Mar 2000

A COMPARATIVE ANALYSIS OF COCKPIT DISPLAY DEVELOPMENT TOOLS

THESIS

Matthew J. Gebhardt, Captain, USAF

AFIT/GE/ENG/00M-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/00M-10

A COMPARATIVE ANALYSIS OF COCKPIT DISPLAY DEVELOPMENT TOOLS

THESIS

Matthew J. Gebhardt, Captain, USAF

AFIT/GE/ENG/00M-10

AFIT/GE/ENG/00M-10

A COMPARATIVE ANALYSIS OF COCKPIT DISPLAY DEVELOPMENT TOOLS

THESIS

Presented to the faculty of the Graduate School of Engineering and Management

of the Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Matthew J. Gebhardt, B.S.

Captain, USAF

March 2000

AFIT/GE/ENG/00M-10

A COMPARATIVE ANALYSIS OF COCKPIT DISPLAY DEVELOPMENT TOOLS

Matthew J. Gebhardt, B.S.
Captain, USAF

Approved:

[illegible]	8 Mar 2000
Chairman, Lt Col Timothy Jacobs	date
Thomas C. Hartrum	8 Mar 2000
Dr. Thomas C. Hartrum	date
[illegible]	8 Mar 2000
Lt Col Mikel Miller	date

Acknowledgements

I would like to express my sincere thanks to my faculty advisor, Lt Col Timothy Jacobs, for his enthusiasm, fresh ideas, and continual guidance and support throughout the course of this thesis effort. His experience and insight were greatly appreciated. I would also like to thank the staff at the Advanced Architecture and Integration Branch of the Information Systems Directorate of the Air Force Research Laboratory for acting as my sponsor in my research effort. Heartfelt thanks go to Mr. Jesse Blair who was instrumental in providing the necessary documentation for the F-22 Raptor. Special thanks goes to Mr. Philip Powers and Mr. Derryl Williams for always being available to answer my questions.

Most importantly, I would like to express my appreciation to my loving wife, Tonya, whose continued support, understanding, and sacrifice has allowed me to make it through the past 18 months of studies. Without her support, my time at AFIT would have been much more difficult. Thanks Babes!

Matthew J. Gebhardt

AFIT/GE/ENG/00M-10

Abstract

Currently, no standard methodology exists that enables cockpit display engineers to evaluate software tools used in the development of graphical cockpit displays. Furthermore, little research has been accomplished in comparing current software development tools with traditional hand-coded methods. This research effort discusses a framework for analyzing cockpit display software development tools and follows through with a detailed analysis comparing today's hand-coding standard, OpenGL, with two of today's cockpit display software development suites, *Virtual Application Prototyping System* (VAPS) and *Display Editor*. The comparison exploits the analysis framework establishing the advantages and disadvantages of the three software development suites. The analysis framework is comprised of several detailed questionnaires that enable the cockpit engineer to quantify important subjective criteria such as learning curve, user interface, readability, portability, extensibility, and maintenance. The questionnaires developed for each subjective criterion contain questions with weighted answers that enable the cockpit engineer to evaluate graphical software development tools. The questions were adapted from multiple sources including personal experience, display experts, pilots, navigators, case tool, and text sources. In addition, the comparative analysis evaluates several objective criteria with respect to development tools and the displays generated with them such as update rate, development time, executable size, and CPU/Memory usage level.

TABLE OF CONTENTS

List of Figures

List of Tables

A COMPARATIVE ANALYSIS OF COCKPIT DISPLAY DEVELOPMENT TOOLS

I. Introduction

1.0 Background

The demand placed on pilots and aircrews has significantly increased with the growing complexities of modern weapon systems and mission parameters. These demands have often exceeded pilot and aircrew abilities and have caused degradation in mission performance and even aircrew fatalities. Projected mission requirements and threat situations in which pilots and aircrews will be involved require vast amounts of operating data. This data can be provided to the pilot and aircrew through a vast network of advanced avionics equipment and supporting ground systems that are essential for the mission to be successful. In addition, the air vehicle, subsystems, and weapons are themselves becoming more highly sophisticated to better support performance goals and extended operating conditions. As a result, the pilots and aircrews are faced with interpreting more data and information, giving more detailed instructions to their onboard equipment, and having less time to perform these functions. The impending result is that the new and improved weapon systems may not yield higher mission success rates if the requirements of the new systems exceed the abilities of the aircrew.

In the past, pilots have been able to function very successfully at the operator level, having direct control over many of the components and subsystems that comprise the weapon system. In this environment, the pilot was able to perform the necessary functions in real-time. Using raw data and information, the pilot was required to monitor, interpret, translate, integrate, and evaluate multiple readouts in order to derive alternatives, decisions and control actions needed to manage the aircraft and perform the required mission. The raw data for these actions was typically obtained from dedicated electromechanical instrumentation in alpha-numeric form. In the current complex environment, this approach is no longer possible due to the ever-increasing amount of data being sent to the pilot. It has become clear that the information processing functions of man can limit the performance of the weapon system. Modern efforts toward cockpit integration are dramatically enhancing the role of the crew, allowing the pilot to effectively exercise appropriate aircraft and mission functions.

At the same time, the advancement of weapon systems with increased complexity has created the technology advances that can be used to solve the problem. Many of the raw data functions performed by the pilot can now be automated offering the pilot more decision-level information for better system management. Mass storage and high speed processing have also provided more and better information than the pilot could have hoped to achieve manually. In addition, these developments have given the pilot greater resources to determine what information is needed and when. Multi-function displays and controls have given the crewstation designer and pilot greater flexibility in the cockpit, allowing the displays to be tailored to pilot or mission requirements. Should the mission need to be changed, the displays can be reconfigured to satisfy the given situation. The ultimate goal of these displays is to increase pilot "situational awareness." The United States Air Force defines situational awareness as "a pilot's continuous perception of self and aircraft in relation to the dynamic environment of flight, threats, and mission, and the capability to forecast, then execute tasks based on the perception" [10:1].

Advancements in technology have allowed the traditional electro-mechanical cockpit instruments to be replaced by cutting-edge graphical displays using cathode ray-tube (CRT) monitors. The use of CRT monitors for these graphical displays has become known as 'glass cockpit' technology. These displays feature enhanced representations of aircraft flight parameters enabling the pilot to make better decisions given the large amounts of incoming data and thereby increasing the pilot's situational awareness.

The pilot's primary perceptions of his environment come through his visual sensory channel. With this in mind, crewstation designers have focused on using graphical formats for displaying critical flight information. The current state of the pilot's environment is represented using a variety of colors, lines, and graphical shapes. In addition to graphical objects, text can be displayed using an assortment of colors to indicate to the pilot a wide range of information including warnings, current aircraft state, and other information. By exploiting the visual sensory channel, crewstation designers give the pilot critical information in a graphical format on a single display. This allows the pilot to focus less on scanning multiple instruments and concentrate more on the information received through a single display source.

Research in glass cockpit technology began as early as 1981. Early on, the focus was on the hardware required to house the displays, not on the information being displayed. Initially, the displayed information was simply a graphical representation of the information already available through the electro-mechanical instruments. Crewstation designers focused on converting the software in the aircraft into a generic graphical format rather than trying to enhance the data for better representation. Little software research was done to enhance the aircraft data to increase pilot situational awareness.

The early focus on hardware display technology led to the problem facing cockpit and display designers today: hardware technology is advancing far more rapidly than software technology. Hardware advancements in recent years have produced smaller computers, consuming less power, and having far more computing capabilities and memory capacity than ever before. These advances are being implemented in the high-performance airborne displays of today's aircraft. The problem is that the software support for these displays is so intricate that they require experts to program them [7:6]. While most initial research efforts focused on the hardware requirements for the glass cockpit, some facilities, such as the Research Triangle Institute (RTI) and the Advanced Architecture and Integration Branch of the Information Systems Directorate of the Air Force Research Laboratory, made attempts at developing software tools allowing display engineers to rapidly develop and reconfigure cockpit displays. RTI developed the first such toolset in the early 1980's called the Interactive Graphics Editor (IGE). Successes with the Interactive Graphics Editor effort by RTI sparked further research by the Advanced Architecture and Integration Branch. Because the IGE was written in the C programming language and aimed at commercial cockpit display design, it did not adhere to government standards requiring the use of the Ada programming language. Teamed with RTI, the Advanced Architecture and Integration Branch began researching a government equivalent to the IGE called the Airborne Graphics Software Support System (AGSSS). Using the Programmer's Hierarchical Interactive Graphics System (PHIGS) as their graphical foundations, both the IGE and the AGSSS allowed an engineer to design a cockpit display from start to finish. These early efforts revealed the need for a software display development tool that could be easily used by an engineer, without the technical ability to program in a higher order language like Ada or C++, to develop dynamic graphical cockpit displays that have the potential to be rapidly reconfigured.

In addition to RTI and the Advanced Architecture and Integration Branch, industry leaders in the graphics arena, such as Silicon Graphics, began developing graphics packages for building graphical applications. One of the graphics packages developed by Silicon Graphics is called the Open Graphics Language, or OpenGL. OpenGL has become very popular in recent years and, due to its popularity among computer graphics software designers, is widely viewed as the graphics standard. In fact, many entertainment companies developing games, movies, and other interactive media use OpenGL as their graphics language.

In the cockpit display environment, OpenGL is also viewed as the standard, but many research facilities require more than what OpenGL can provide. Facilities like the Advanced Architecture and Integration Branch are looking for a software graphics package/tool that can be easily used by the non-programming cockpit engineer. OpenGL, while a good graphics foundation, has an extremely limited user-interface and, with its required knowledge of C or C++, has a relatively steep learning curve. Research facilities are looking for a software tool with a graphical user interface (GUI) for developing cockpit displays that are easily developed and rapidly reconfigurable.

In an effort to satisfy these facilities, several companies have developed software tools that enhance the creation of graphical cockpit displays. One such company, Virtual Prototypes, Inc. (VPI), has developed several tools that are used to build graphical displays. Using tools such as VPI's *Virtual Application Prototyping System* (VAPS) and VPI's *C-Code Generator* (CCG) the designer can create the displays using a point-and-click environment by drawing squares, circles, text, and other shapes. The designer can then animate the display, connect it to an outside application, or immerse it in a hardware-in-the-loop simulation environment. By taking advantage of advancing PC technology and the development of Windows NT, VPI has created a software tool for rapid display development and implementation.

In addition to VPI, another company, SCS Engineering, Inc., has developed a software tool that can also be used to build graphical cockpit displays. SCS's *Display Editor* tool uses OpenGL as its foundation, but allows the engineer to design the "look" of the display in a point-and-click environment similar to that of VAPS. Once the display is drawn, the development team at SCS Engineering accomplishes the necessary animation. Future releases will enable the display designer to fully define and implement the display dynamics themselves.

This research effort will look at the tools developed by Virtual Prototypes and SCS Engineering and compare them to the unofficial graphics standard of today, OpenGL.

1.1 Display Development Process

Several problems face cockpit display designers. First, the requirements of the displays must be analyzed. The designers must determine the performance requirements such as update rate, CPU usage, and memory usage. The displays must also satisfy graphical requirements such as the size of the display, size of the objects within the display, and the font sizes. The requirements are typically summarized in a standards document forming a foundation for future display development.

The next problem facing cockpit display designers is hardware configuration. Depending on the development environment, there may be several different hardware configurations available. In the simulation environment alone, there are countless hardware configurations that range from high-end workstations with multiple processors to a simple, single-processor, personal computer (PC) running Microsoft Windows (NT/95/98). The hardware that will be used for display development may be something the display designer already has or they may want to purchase new hardware specifically for developing and running the displays. In simulation environments, like the Integrated Test Bed facility at the Advanced Architecture and Integration Branch, PC platforms act as both display development station and simulation platform.

The final problem, and potentially the biggest decision, is the choice of software graphics package to employ for display development. There is a broad range of graphics packages available on the market today. OpenGL, VAPS, and *Display Editor* are only three of these; however, these three graphics packages represent a solid cross section of the toolsets available. OpenGL is widely used but was not engineered specifically with cockpit displays in mind. VAPS was engineered as a human-machine interface (HMI) development tool (cockpit displays are only one type of HMI). And finally, Display Editor is being developed specifically for use in designing cockpit displays.

1.2 Problem Statement

Design, develop, and test two dynamic graphical displays using OpenGL, VAPS, and Display Editor. Show that each development tool set satisfies the display requirements for look and performance. Accomplish a comparative analysis of the three tools by immersing the displays developed with each tool in a simulation environment and executing the displays across several hardware platforms, revealing the advantages and disadvantages of each.

1.3 Summary of Approach

The problem statement mentioned in Section 1.1 contains several objectives that need to be accomplished. The first and foremost objective is developing the display requirements and standards. These requirements will potentially come from several different sources. Requirements from aircraft developers, such as Lockheed-Martin and Boeing for the F-22, include such things as display layout, memory and CPU limitations, and color. Additional requirements may come from simulation facilities, such as the

Advanced Architecture and Integration Branch, which include items like real-time performance. The graphics package chosen for display development must satisfy these display requirements. In addition, cockpit display designers have user-interface, code maintenance, portability and other important requirements that must be met by the development tool set. These requirements and their development will be discussed in later chapters.

The second objective from the problem statement is to design and develop the displays. For this research effort, two displays are developed: the Attitude Director Indicator (ADI) and the Horizontal Situation Indicator (HSI) from the upcoming F-22 Raptor cockpit. These displays were chosen primarily for their dynamic characteristics and requirements. The dynamic characteristics of the ADI and HSI test the ability of the graphics package with which the display was developed. Both the ADI and HSI must meet *real-time* performance requirements. For the purpose of this research, *real-time* performance will be defined as:

- **Real-time performance: A minimum data update rate of 16Hz with a desire for 30Hz or greater [6:11].**

The real-time performance requirement leads us to the next objective. According to the problem statement, the displays are immersed in a simulation environment. This tests the ability of the displays to meet the display requirements using real aircraft data. The simulation environment for this research is the *Re-configurable Avionics Modeling* *and Simulation System* (RAMSS), developed by SCS Engineering Inc. The displays and the simulation system run on a PC platform running Microsoft Windows NT.

The final objective is the cornerstone for this research effort. Once the displays are developed and tested to ensure display requirements are met, the tools used to build the displays are analyzed. This comparative analysis comprises all aspects of cockpit display development such as toolset cost, display lifecycle, and display performance. Several questionnaires and tests are used to analyze subjective and objective characteristics about each development tool set. These questionnaires contain weighted questions that reveal the advantages and disadvantages of each development tool set. Part of the final objective, but a separate experiment altogether, is executing the displays using different hardware configurations. This experiment reveals important characteristics of each development tool set in different hardware environments.

The first three objectives are building blocks for the final objective. The comparative analysis of the display development toolsets reveals characteristics that are common with graphics applications. The questionnaires and tests, while applied only to OpenGL, *VAPS*, and *Display Editor* for this research, can also be applied to other display development graphics packages. The goal is not to establish the best tool, but instead to provide a framework for analyzing display development graphics packages.

1.4 Scope of Research Effort

This research effort will not attempt to design, develop, nor test every possible cockpit display that may be used in today's aircraft or aircraft of the future. Instead, it focuses on the two displays mentioned above, the ADI and HSI, from the next generation F-22 Raptor cockpit. The ADI and HSI displays are designed using OpenGL, *VAPS*, and the *Display Editor* software toolset in accordance with standards established for the F-22 cockpit displays. These standards have been in development for several years and continue to change to adapt to new threat situations. The standards document used for display development for this research effort is titled *F-22 Air Vehicle Cockpit Design Description Document* and dated July 1996.

1.5 Research Assumptions and Limitations

There are a number of assumptions that must be made when accomplishing the research objectives discussed above. The first is that cost can be a limiting factor to success in this research area. The OpenGL tool set is free to the public through the Silicon Graphics website (http://www.sgi.com) while the other two development tools incur license costs. VAPS and Display Editor were chosen because they were readily available through the Advanced Architecture and Integration Branch at no cost.

The Advanced Architecture and Integration Branch at the Air Force Research Lab Information Systems Directorate will provide all necessary hardware and software. The hardware and software required consist of current OpenGL software libraries, the VAPS toolset, the Display Editor toolset, several Windows NT platforms (with varying hardware configurations), up to date versions of Microsoft Visual C++, and all appropriate licenses. Finally, the Advanced Architecture and Integration Branch will provide all necessary display standards documents from the F-22 design team at Lockheed-Martin and Boeing.

The comparative analysis answers several questions about the three specific graphics packages used in this research. In addition, this research provides a foundation for future analysis or comparisons of graphics packages. This research effort does not answer every potential question about OpenGL, VAPS, or Display Editor. On a further note, having some general knowledge about real-time simulation and cockpit displays facilitates understanding. The questionnaires and tests used in the analysis for this research can easily be adapted and applied to any generic graphics package for evaluation.

1.6 Document Overview

This document contains six chapters. Following this introductory chapter is a chapter dedicated to familiarizing the reader with OpenGL, VAPS, and Display Editor. In addition, the second chapter examines the past efforts of the Research Triangle Institute and the Advanced Architecture and Integration Branch. The third chapter details the objectives from Section 1.2 while Chapter 4 discusses the display design methodology and the displays developed for this research effort. Chapter 5 presents a detailed discussion about the questionnaires and tests used in the comparative analysis of the three development toolsets. The next chapter, Chapter 6 presents the results and an analysis of the questionnaire and test results and the final chapter consists of a summary, conclusions, and potential research for the future.

II. Literature Review and Development Tool Description

2.0 Historical Perspectives – Why the Glass Cockpit

Advancing technologies in hardware have led to many changes in the traditional cockpit. Computer technology has advanced to the point that the computers are much faster, can compute more complex routines, and are small enough to be placed onboard the aircraft. These computers have enabled the aircraft to automate many of the raw data functions that the pilot had done in the past. By using the computers in this way, the pilot can be presented with more decision level information rather than raw data [14:15]. Still, however, the pilot is forced to scan multiple instruments in order to gather critical information. Gathering information from these instruments may only take seconds, but these seconds may mean life or death for the pilot and aircrew [5:1-2].

The glass cockpit concept was born out of efforts to put graphical displays in the cockpit presenting more critical information on a single display, saving the pilot those precious seconds it may take to read multiple displays. In addition, information can be displayed in a more visual format (i.e. colored objects and lines instead of conventional gray scale text and symbols), resulting in better decisions. Software technology has advanced at a slower rate than hardware technology making these graphical displays time-consuming and costly to develop because programming experts must code them by hand. The Air Force and industry need tools to develop cockpit displays quickly, accurately, and inexpensively.

The reason original displays were so expensive and time consuming to develop was because they required an expert to program them. The programming was time intensive and laborious. Furthermore, if a problem was found, whether it was syntactical or semantic, it required the display to be reworked, costing even more time and money. Finally, during the lifetime of the display, should it need to be added to or modified, more often than not, it had to be entirely re-accomplished.

For these reasons, several companies and Air Force facilities began developing software development tools for efficient creation of cockpit displays. Initial development efforts, like that of the Research Triangle Institute and the Advanced Architecture and Integration Branch, used the PHIGS graphics package. Their research provided a stepping-stone for the research being accomplished today in the cockpit simulation field.

Silicon Graphics introduced the OpenGL software package in 1992 and it has been adopted widely throughout the graphics industry. Virtual Prototypes, Inc. began developing their *VAPS* software toolset in 1987. *VAPS*, dedicated to human-machine interfacing, is being used by many Air Force facilities as well as some aircraft developers like Lockheed-Martin and Boeing. In conjunction with the Advanced Architecture and Integration Branch, SCS Engineering, Inc., began developing their *Display Editor* toolset in 1998. While based on the OpenGL graphics package, *Display Editor* is designed specifically for developing cockpit displays.

2.1 The Interactive Graphics Editor

In the mid 1980's, the Research Triangle Institute began developing a software development toolset that could support the design and creation of 2D and 3D cockpit displays while at the same time reducing the lifecycle costs of display development. This toolset, called the Interactive Graphics Editor (IGE), was written in the C programming language. The IGE supported the rapid prototyping of 2D and 3D cockpit display formats, allowed a preview of their animation, and generated the source code for the animation automatically [7:441].

The IGE is a hardware and software system supporting rapid development, testing, and evaluation of cockpit display formats and the automated generation of source code. Using IGE, a display designer can define cockpit displays without resorting to conventional programming methods [7:441]. Developed for NASA, its primary goal was to allow creation of cockpit displays pictorially rather than procedurally. Figure 1 shows a block diagram of the *Interactive Graphics Editor* system.

The IGE consists of a PC-based front-end workstation and a display system consisting of a MicroVAX II host computer and an Adage 3000 Programmable Display Generator (PDG). The PC-based workstation controls all of the system inputs through the use of the various input devices in Figure 1. The primary input device for the system is the data tablet. With the tablet, the display designer can draw and define the various objects within the display. Using the IGE menu-based interaction mechanisms, the display designer can define the dynamic characteristics of objects within the display.

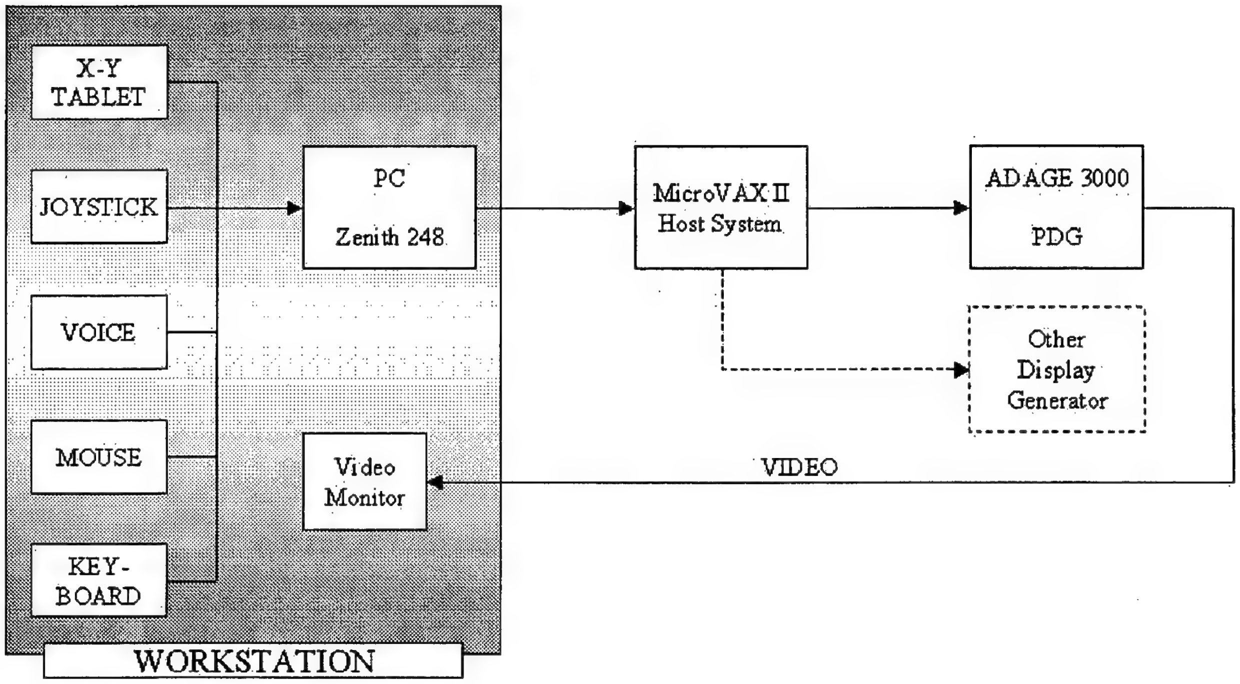


Figure 1. Block Diagram of *Interactive Graphics Editor*

The IGE was written in the C programming language using graphics bindings from the Programmer's Hierarchical Interactive Graphics System (PHIGS) language and was intended to be a commercial product. The government, specifically the U.S. Air Force, needed a toolset that used the Ada programming language since Ada was the underlying standard for all Department of Defense technology. This led RTI, teamed with the Advanced Architecture and Integration Branch, to begin research in an Ada-based graphics editor toolset.

2.2 Airborne Graphics Software Support System (AGSSS)

The *Airborne Graphics Software Support System* (AGSSS) had its origin at Wright-Patterson AFB, OH in the early 1980's. Developed by the Research Triangle Institute and the then named Wright Laboratory (today the Air Force Research Laboratory), the AGSSS software tool helped the engineer to design a cockpit display, define its dynamic characteristics, and generate the code required to animate the display without knowledge of any higher-level programming language, such as Ada or C++.

The software was modular in nature allowing the designer to test, modify, and generate code throughout the development cycle. It was engineered using a Windows-based environment with a point-and-click type of interface adapted from RTI's Interactive Graphics Editor interface. The AGSSS software tool was developed using the government standard Ada programming language and the bindings of the Programmer's Hierarchical Interactive Graphics System (PHIGS) graphics language. The engineer designs the display in one window and can see the code generated in another window. The generated graphics code is then sent through a code converter to convert it into Ada/PHIGS runtime code. The generated code could then run as a graphical display using dedicated Avionics hardware. Figure 2 shows a block diagram of AGSSS and Figure 3 shows an example display generated using the AGSSS development toolset.

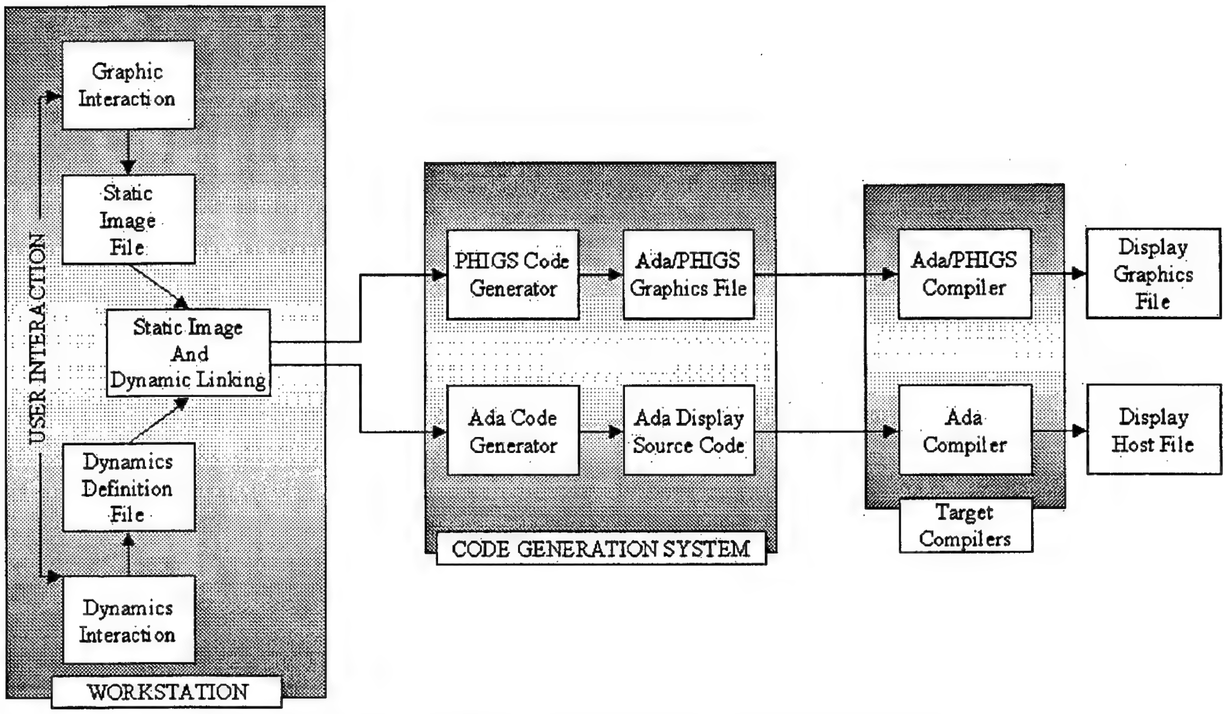


Figure 2. Block Diagram of the AGSSS Display Generator

The display designer, using the workstation in Figure 2, defines the dynamic and static portions of the display. These image files are then sent through the code-generators creating two source files. The first file is a graphics file defining the graphical image and the other is the Ada source code that defines the dynamic movement of the display. These source files can then be compiled, hosted, and executed on appropriate hardware platforms.

The AGSSS and RTI display development environments demonstrated the ability to build graphical cockpit displays without the need of technical programming knowledge. Engineers lacking coding experience could use the tool to develop integrated graphical displays that were dynamic in nature. These displays could also be reconfigured for different missions with limited down time.

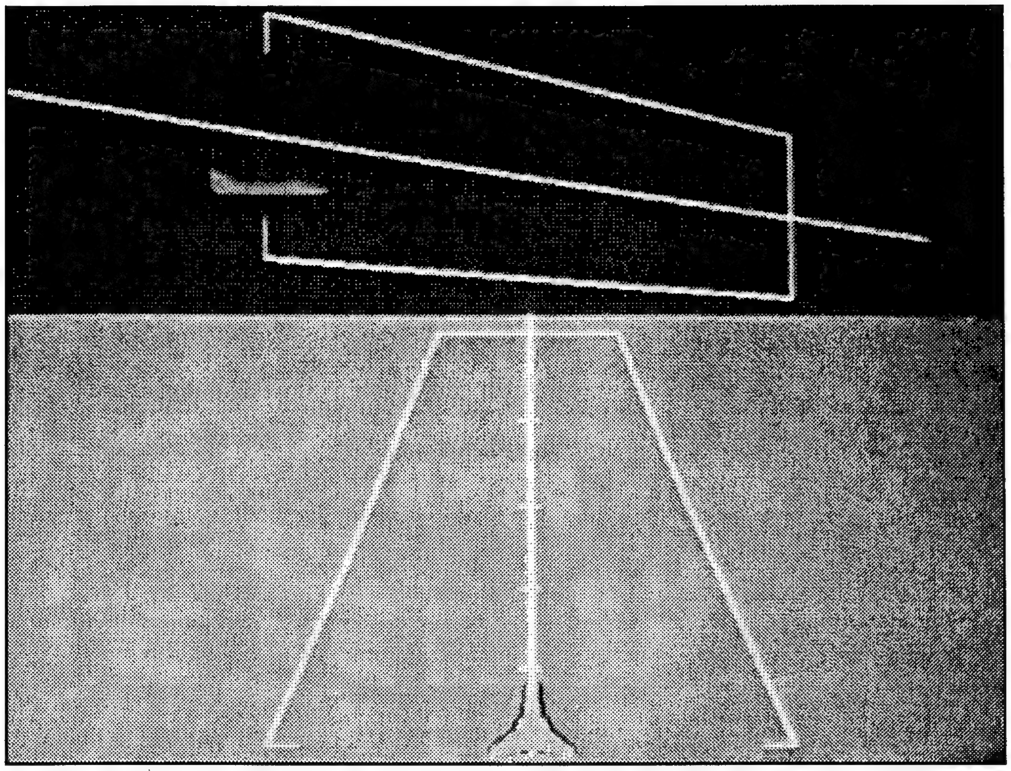


Figure 3. Example Instrument Landing Display Generated Using AGSSS

In fact, as revealed in the AGSSS final report, the development tool increased productivity by a factor of 10 over that obtained through conventional display development methods (e.g. hand-coding the displays in Ada/PHIGS) at the Advanced Architecture and Integration Branch [7:3]. The report further states that even greater benefits could come with fine-tuning of the AGSSS software. The research accomplished on the AGSSS program proved that a software display development tool was beneficial and cost effective.

The AGSSS tool, although a moderate breakthrough in its own right, had its drawbacks as well. Though it cut down the development cycle in both cost and time, it was highly hardware and software dependent. Dr. Jorge Montoya in *AGSSS: The Airborne Graphics Software Support System* describes these dependencies. He explains that the AGSSS tool required the use of MicroVAX III workstations and ADAGE 3000 graphics processors. He also states that AGSSS required Ada and PHIGS compiler licenses for the PC's, MicroVAX's, and ADAGE graphics computers used throughout the display development lifecycle. Though the AGSSS boasted a refresh rate of 30-Hz, Montoya explains, this was only accomplished using several ADAGE 3000 graphics processors and a dedicated cockpit display interface [6].

Even though the AGSSS was a breakthrough in its time, hardware and software have continued to advance in recent years, driving the need for a more advanced cockpit display development tool. This need is being researched and met by commercial companies who are developing new software tools to carry on what the AGSSS started.

2.3 The Open Graphics Language (OpenGL)

The Open Graphics Language was developed by the Silicon Graphics Corporation and was first released in 1992. The Open Graphics Language, otherwise known as OpenGL, is an applications programming interface (API) supporting the creation and rendering of high-performance 2D and 3D graphics applications. An API is a standard library for developing applications software and as such, API's typically simplify the software development process and reduce development cost and time. OpenGL is a platform independent API, with the ability to be written once and deployed on multiple platforms [15: 2-3].

OpenGL has language bindings in many higher-order programming languages including C, C++, Fortran, Ada, and Java [8:2]. It allows the rendering of simple geometric primitives like points, lines, or polygons and highly complex shaded and texture-mapped curved surfaces. Software developers can manipulate geometric primitives, create display lists, and use OpenGL bindings for model transformations, lighting, texturing, anti-aliasing, blending, and other functions. This research effort uses the C++ programming language and common OpenGL functions for anti-aliasing, blending, and transformations to accomplish cockpit display dynamics. The displays created for this research are 2-dimensional and use display lists when appropriate.

The OpenGL API is available as freeware on several World Wide Web sites, including the Silicon Graphics website (http://www.sgi.com). Licensing fees apply only when the OpenGL source code is desired [8:5]. OpenGL runs on every major operating system including Mac, OS/2, UNIX, Windows 95, Windows NT, Linux, OPENStep, Python, and BeOS [9:3]. OpenGL also works with every major windowing system, including Presentation Manager, Win32, and X-Window System. This research effort focuses on the Microsoft Windows NT version of the OpenGL Libraries and header files using the C++ programming language as mentioned previously.

OpenGL is designed using a pipelined architecture. Figure 4 shows the general pipeline that OpenGL follows. As the figure depicts, the geometric vertices defined in the application are unpacked. The operations on the vertices are then accomplished and converted into screen geometry to be sent to the frame buffer for displaying. If display lists are used, as this research does at times, the vertex operations are saved in a cache-type buffer for later reuse. The use of display lists increases performance of applications by reducing the number of vertex operations performed during runtime.

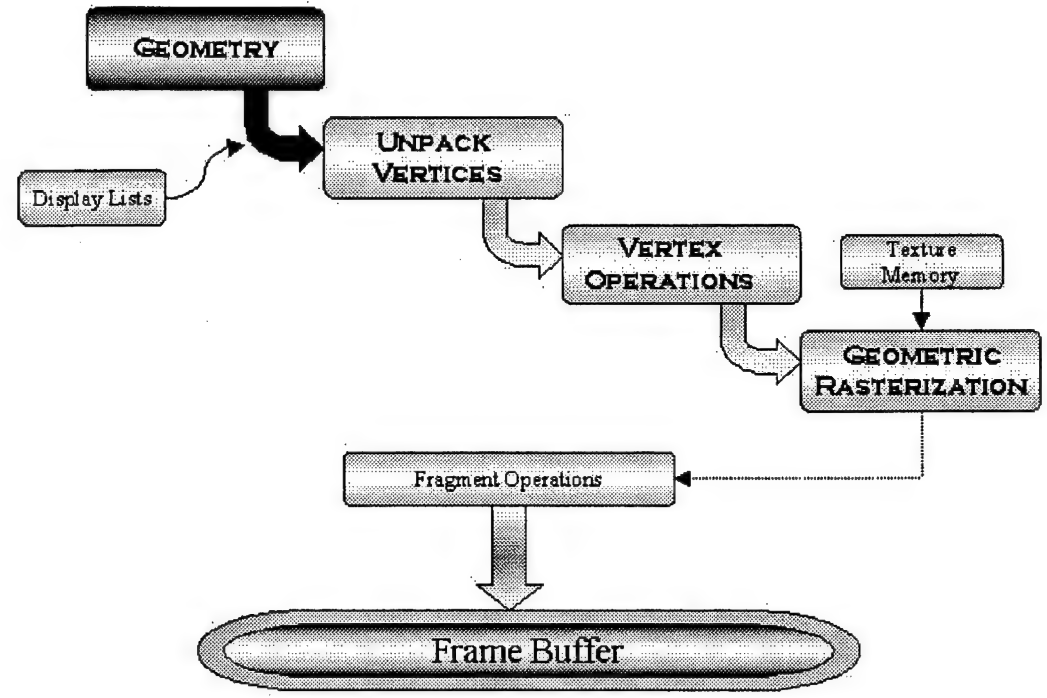


Figure 4. The OpenGL Graphics Pipeline [9:3]

This research effort focuses on cockpit displays using 2-dimensional representations of objects, lines, and primitive polygons. Using the C++ bindings provided by OpenGL, displays similar to that in Figure 5 can be produced. The display in Figure 5 is a simple bird's-eye view of the aircraft that depicts aircraft heading and steering point. Though this display is quite basic, it shows that OpenGL has the ability to generate dynamic cockpit displays.

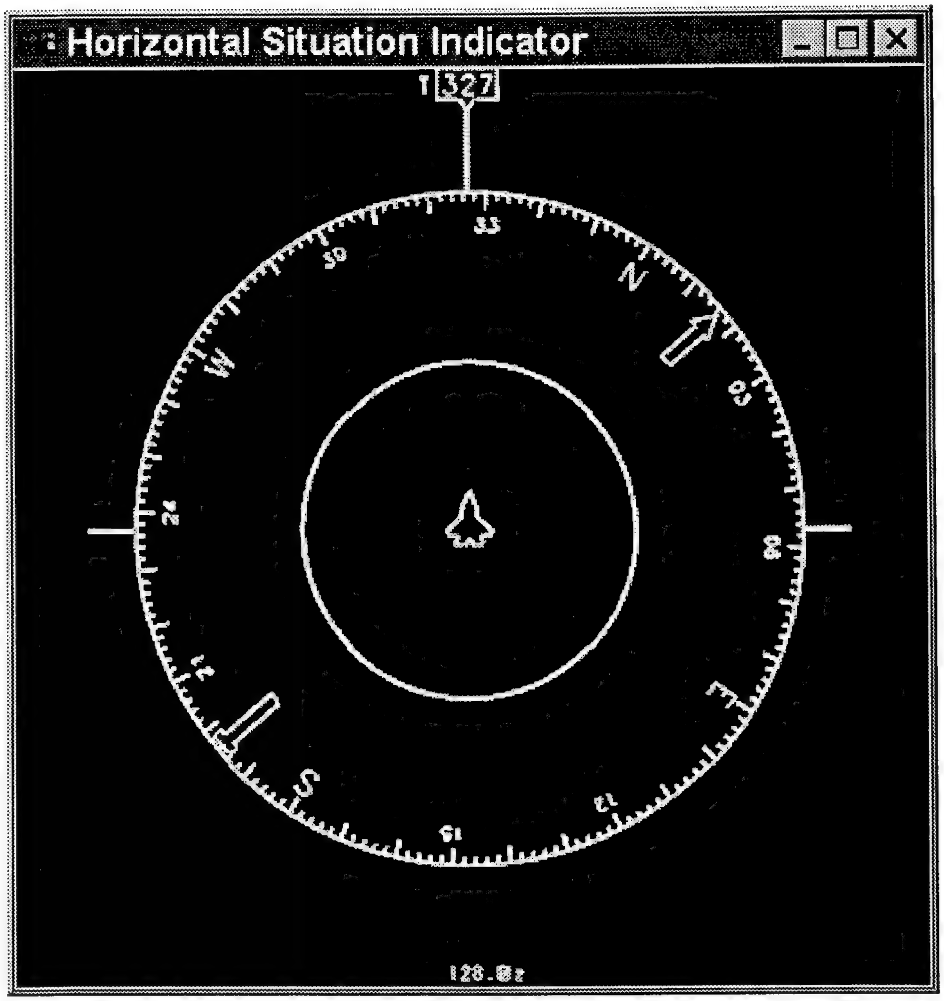


Figure 5. A simple Heading Indicator accomplished in OpenGL

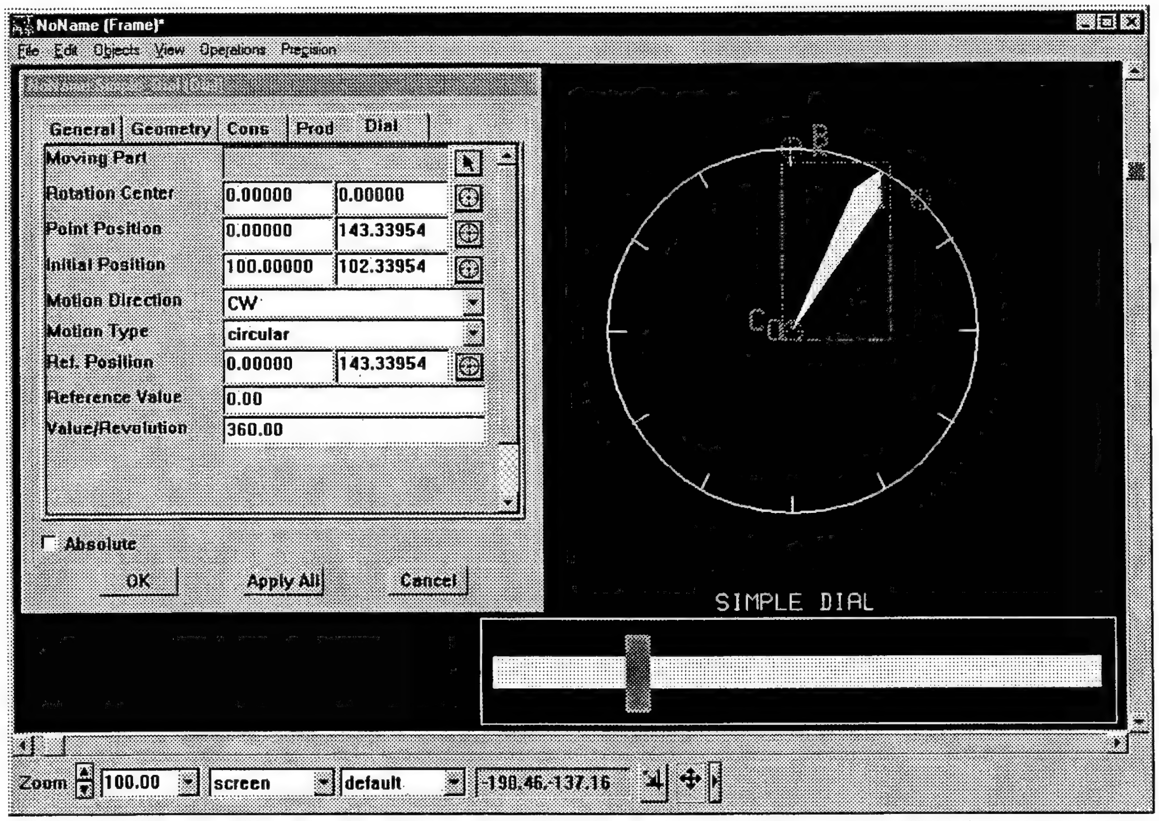
2.4 Virtual Application Prototyping System

In 1987, Virtual Prototypes, Inc. (VPI), out of Montreal, Canada, began researching the development of a software tool environment in which the display engineer could design, animate, and integrate a display from start to finish [2]. Of course, this could be done using a graphics package like PHIGS or OpenGL, as discussed above; however, VPI wanted to add one critical design feature, a simple point-and-click graphical user interface (GUI). With a GUI in a windows environment, the display could be designed with little or no knowledge of higher-level languages like C++. VPI introduced their *Virtual Applications Prototyping System* (VAPS) in 1990.

Actually, VAPS is three individual tools rolled into a single development environment. The three tools are the Object Editor, the Stateforms Editor, and the Runtime Environment [12:6]. Breaking each stage of development into three separate tools, VAPS facilitates easy modifications and changes to the display at any level of display development.

The first stage of development is determining the look and feel of the display. Typically, the general display design comes from standards documents and display requirements, as is the case with this research effort. However, if standards and requirements are not immediately available, the displays can be designed in a general format that allows details to be added later. Once the general display design is decided upon, the VAPS Object Editor (OE) can then be used to draw the display. The OE is a window application using a point-and-click format. The OE allows the designer to draw graphical primitives, such as lines, points and polygons (including text), and combine them in such a way to generate the look and feel of the cockpit display. All of the objects drawn within OE are selectable and groupable. Grouping objects together is one important feature of the VAPS OE. In fact, the VAPS OE contains several predefined objects that the designer can use to group primitives to form dials, switches, potentiometers, and other devices that are commonly found in the human-machine interface environment. Once grouped, the display designer must define the dynamic characteristics of the object. Which object(s) within the group will move, where the initial position of the grouped object is located, and the range of motion of the moving object(s) are just a few of the characteristics that must be defined. Figure 6 shows a simple dial defined using the Object Editor and the dial properties window showing the dial object definitions.

Each grouped object also has several "plugs" that allow the object to consume and produce data. Within the VAPS OE, the display designer must define a file called a channel [1:4]. This channel contains variables that hold data values either produced or consumed by the VAPS objects within the display. The OE allows the designer to connect input sources within the display to the corresponding output sources using



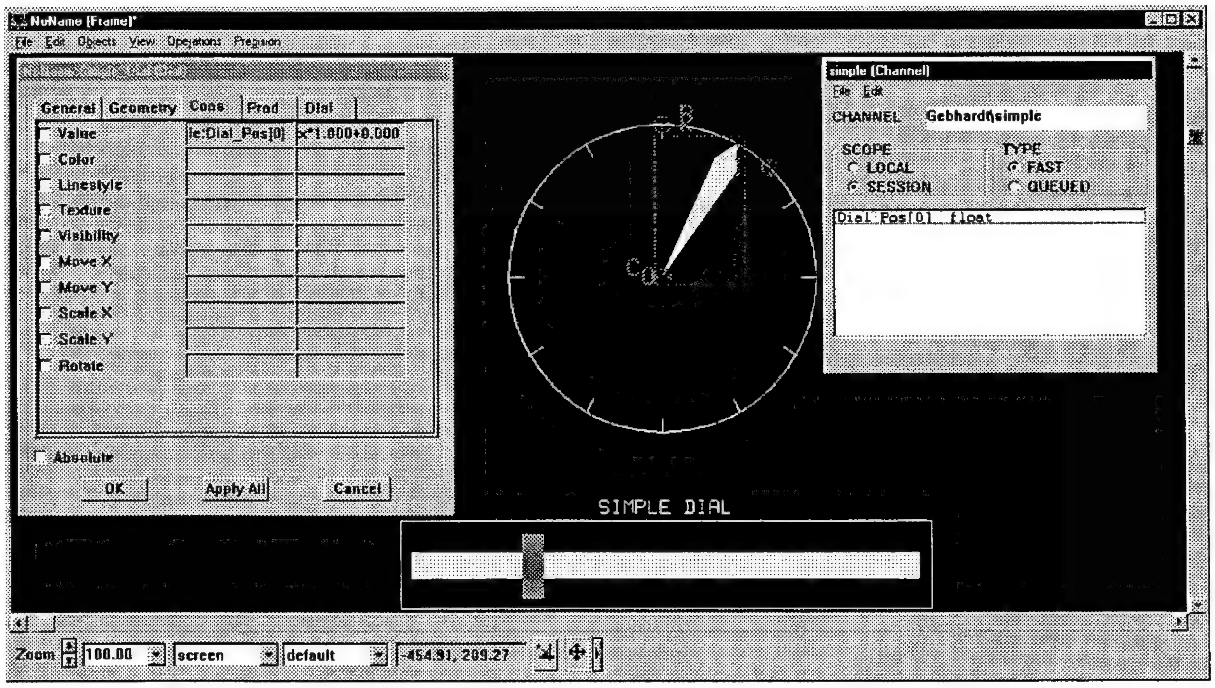
Moving Part		
Rotation Center	0.00000	0.00000
Point Position	0.00000	143.33954
Initial Position	100.00000	102.33954
Motion Direction	CW	
Motion Type	circular	
Ref. Position	0.00000	143.33954
Reference Value	0.00	
Value/Revolution	360.00	

Zoom 100.00 screen default -198.46,-137.16

Figure 6. VAPS Object Editor Simple Dial with Properties Window

variables within the channel file. Consider the simple dial and potentiometer seen in Figure 6. When the potentiometer is moved with the mouse, the pointer on the dial rotates. The potentiometer object writes data to a producer plug while the dial reads data from a corresponding consumer plug, updating its location accordingly. Figure 7 shows the dial consumer plug and the channel file variable to which it is connected.

The potentiometer produces data that is stored in a shared memory location (channel file variable) pointed to by the corresponding dial consumer plug. Figure 7 reveals only one side of the connection (consumer to channel). The other side of the connection is identical, except it connects a channel file variable to a producer plug. Without these connections, the VAPS objects would not be able to communicate with each other. In fact, the channel file provides the mechanism for the VAPS created display



Value	le:Dial_Pos[0]	x*1.000+0.000
Color		
Linestyle		
Texture		
Visibility		
Move X		
Move Y		
Scale X		
Scale Y		
Rotate		

Absolute

CHANNEL Gebhard\simple

SCOPE: LOCAL [✓ SESSION]

TYPE: [✓ FAST] QUEUED

```
Dial_Pos[0]  float
```

100.00 screen default -454.91, 209.27

Figure 7. VAPS Object Editor Plug Connections and Channel File

to communicate with an outside application, as accomplished in this research effort.

After all of the appropriate plugs have been connected, the designer develops a finite state machine that describes the state of the interface at any given moment throughout the lifecycle of the display. Put simply, should the display have several different modes (which VAPS calls “frames”), the state machine definition will determine which frame to display. Changes of state are defined within a small program, written in C or C++, and are based on a conditional check, which is determined by the designer. For this research, the two displays developed, the Attitude Director Indicator (ADI) and Horizontal Situation Indicator (HSI), have only one operational frame. A state machine definition is used to initialize some portions of the display. For example, with the ADI, the state definition initialized a variable defining the time of day. This variable controls certain colors that are loaded when the display is executed.

With the state machine defined, the designer uses the VAPS Runtime Environment (RE) to analyze and test the display. The RE allows the designer to open and run the display. Testing and analysis can be accomplished at this level to determine that the display looks properly, contains the appropriate data types, and interacts properly among its elements (i.e. the potentiometer in Figures 6 and 7 causes the dial to rotate properly).

So far, VAPS has been discussed in a stand-alone mode of operation without interaction with outside applications like a cockpit simulation. The displays for this research effort demand data from an aircraft simulation environment. In order to accomplish this, VAPS provides the above mentioned channel file and several communications subroutines. A small application written in C or C++ can pass data through the channel file to the VAPS objects. Any consumer plugs connected to that channel file member will read the data and update accordingly. Seen in Figure 8, a VAPS object can be driven with another VAPS object or with a simulation system.

For example, instead of using the potentiometer to drive the dial (Method 1 in Figure 8), a C++ application could be written to send data to the appropriate channel file variable consumed by the dial (Method 2 in Figure 8). The C++ application acts as the simulation and produces data, which is sent to the channel file. The dial can consume the data from the same channel location as before. Using this capability, the display can be immersed in a cockpit simulation environment reflecting appropriate aircraft data.

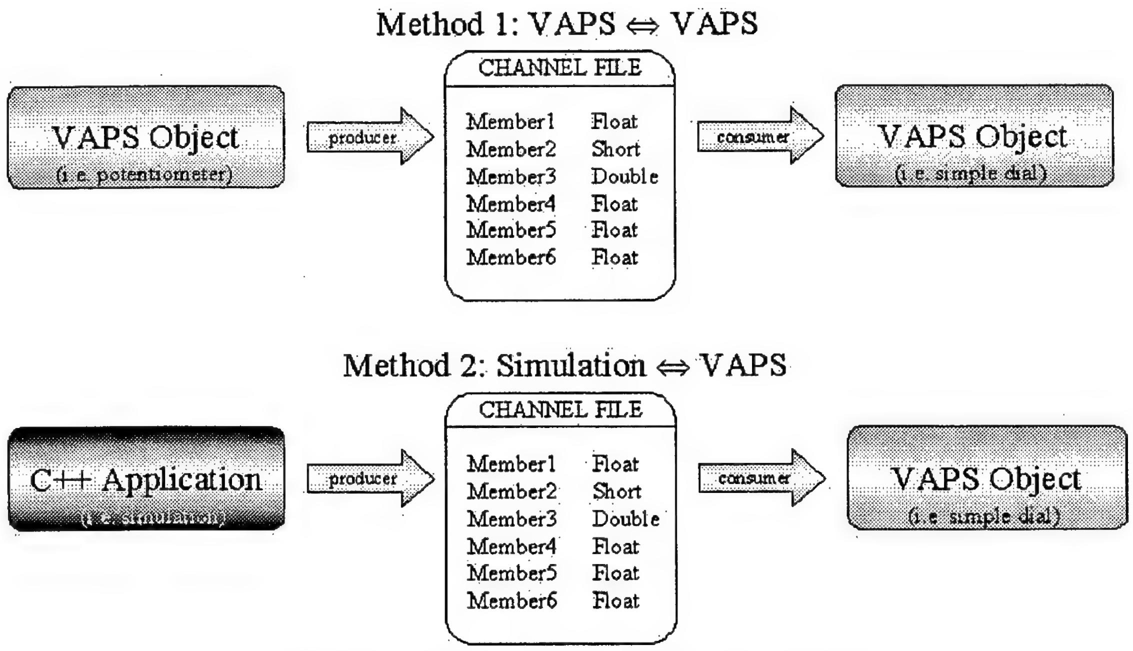


Figure 8. Driving a VAPS Object with another VAPS object or with a C++ Application

2.5 SCS Engineering's Display Editor

In 1997, the Advanced Architecture and Integration Branch awarded a Small Business Innovative Research contract to SCS Engineering, Inc. out of Los Angeles, CA. Under this contract, SCS Engineering would provide several software and hardware simulation packages. The primary focus of the contract was to replace the legacy simulation host in the Advanced Architecture and Integration Branch's simulation facility. In addition to this replacement, the contract called for SCS Engineering to deliver several Windows NT-based simulation stations under the name Reconfigurable Avionics Modeling and Simulation System. These simulation stations run real-time aerodynamic models for the A-7, F-15, and C-130 created by SCS Engineering, Inc. under a Small Business Innovative Research (SBIR) contract.

As part of the final delivery, SCS Engineering was contracted to deliver a cockpit display development tool. Display Editor was the result of this contract requirement. Due to the timing of the contract, the Display Editor toolset has not yet been released to the public and remains in Beta testing. The toolset is designed to use a point-and-click window-based environment as seen in Figure 9. Currently, the display designer develops a display using the windowed environment of Figure 9. Once all display objects and text are created, the display is saved in its graphical format and run through a code generator provided with the Display Editor toolset. The code generator creates a file (*.ogl file) with OpenGL definitions for all objects and text within the created display. A sample display generated by the Display Editor toolset is seen in Figure 10.

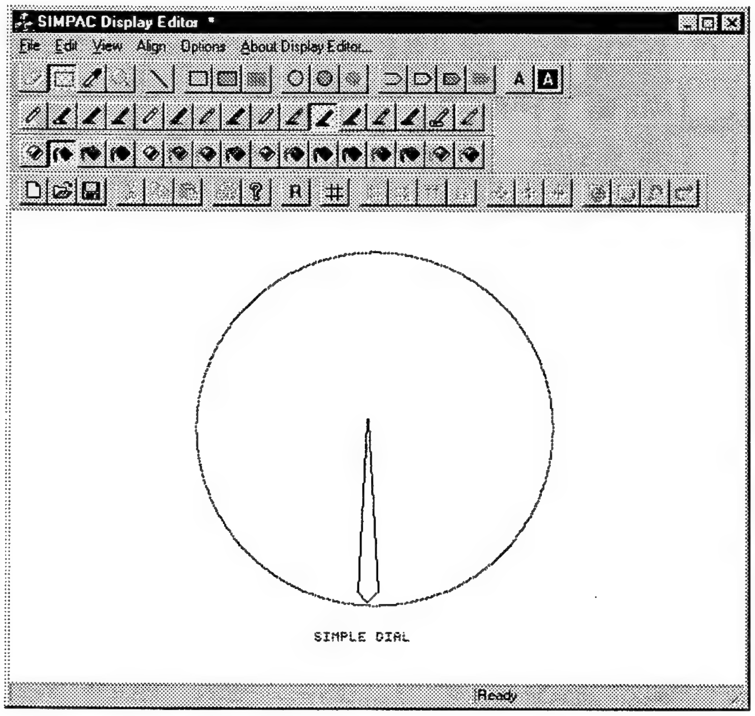


Figure 9. Display Editor sample display of a simple dial.

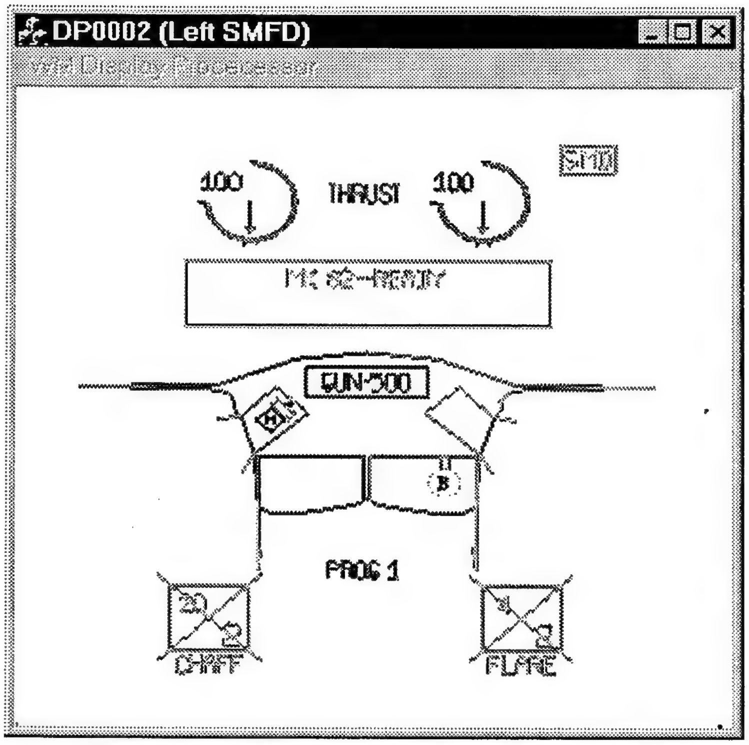


Figure 10. Stores-Management Display Created with Display Editor

Once the code has been generated, the display designer has one of two options. They could define the dynamics of the display themselves by hand-coding them using OpenGL bindings or they could send the OpenGL file to SCS Engineering for dynamic definition, providing a detailed description of the animation of the display as well. As mentioned above, this is the current configuration of the Display Editor toolset. Future versions may allow the display designer to develop the display and define its dynamic functionality using the point-and-click windowed environment.

III. Experimental Approach

3.0 Introduction

The approach taken for this research was briefly introduced in Chapter 1. This chapter goes into more detail about the approach, giving an in-depth description of each phase of research. This research effort will not attempt to determine which of the development toolsets is better. Instead, this research focuses on the evaluation techniques of cockpit display development software and the displays built using OpenGL, VAPS, and Display Editor. As introduced in Chapter 1, there are 5 primary phases for this research.

- I. **Develop cockpit display requirements and standards.**
- II. **Design and develop the two cockpit displays, the Attitude Director Indicator (ADI) and the Horizontal Situation Indicator (HSI) using OpenGL, VAPS, and Display Editor.**
- III. **Immerse cockpit displays (ADI and HSI) in a simulation environment ensuring display requirements and standards are met including any real-time performance requirements.**
- IV. **Perform analysis of development tools determining advantages and disadvantages of each using subjective and objective evaluation techniques.**
- V. **Test resulting displays across several hardware platforms.**

The last two phases are the critical part of this research. The first three phases must be accomplished in order to perform the last two. Each phase builds on the previous one, culminating with the final analysis and platform testing. The resulting analysis will help engineers make better decisions about hardware and software purchases for their cockpit simulation environments. The next six sections go into greater detail about each of the phases of research with the final section acting as a summary.

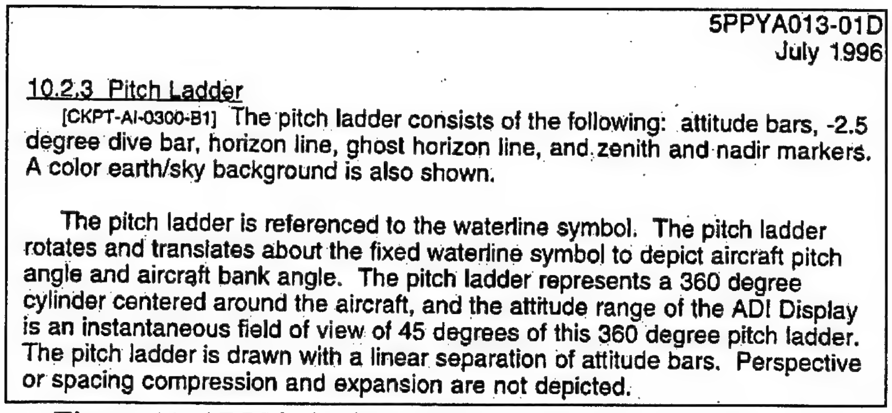
3.1 Developing Display Requirements and Standards.

Developing display requirements and standards can be extremely time consuming. For a typical operational aircraft, like the F-22 Raptor, standards development may take months or even years to accomplish. However, U.S. Air Force engineers and aircraft contracting firms have already accomplished the standards and requirements development work for this research. Briefly mentioned in Chapter 1, the cockpit displays used in this research effort come from the cockpit design of the F-22 fighter being developed by Lockheed-Martin and Boeing [4]. For the past several years, these two companies have been researching all aspects of the aircraft including the cockpit layout and displays. The exact look and feel of each display has gone through several cycles of design.

This research effort uses the *F-22 Air Vehicle Cockpit Design Description Document* dated July 1996 [4]. This document, coupled with several appendices, describes the exact location, dynamic motion, and appearance of object and symbols within each cockpit display. For example, Figure 11 contains a detailed textual description of the pitch lines for the Attitude Director Indicator (ADI) as described in the *F-22 Air Vehicle Cockpit Design Description Document*.

In addition to the textual description, each display has a related appendix that goes into pixel-by-pixel descriptions of the symbology located on that display. Figure 12 shows the graphical description that goes along with the textual description seen in

Figure 11. For this research effort, the requirements and standards development was simple because the personnel at Lockheed-Martin and Boeing had already accomplished this work.



5PPYA013-01D
July 1996

10.2.3 Pitch Ladder

[CKPT-AI-0300-B1] The pitch ladder consists of the following: attitude bars, -2.5 degree dive bar, horizon line, ghost horizon line, and zenith and nadir markers. A color earth/sky background is also shown.

The pitch ladder is referenced to the waterline symbol. The pitch ladder rotates and translates about the fixed waterline symbol to depict aircraft pitch angle and aircraft bank angle. The pitch ladder represents a 360 degree cylinder centered around the aircraft, and the attitude range of the ADI Display is an instantaneous field of view of 45 degrees of this 360 degree pitch ladder. The pitch ladder is drawn with a linear separation of attitude bars. Perspective or spacing compression and expansion are not depicted.

Figure 11. ADI Pitch Line Textual Description [4:Section 10]

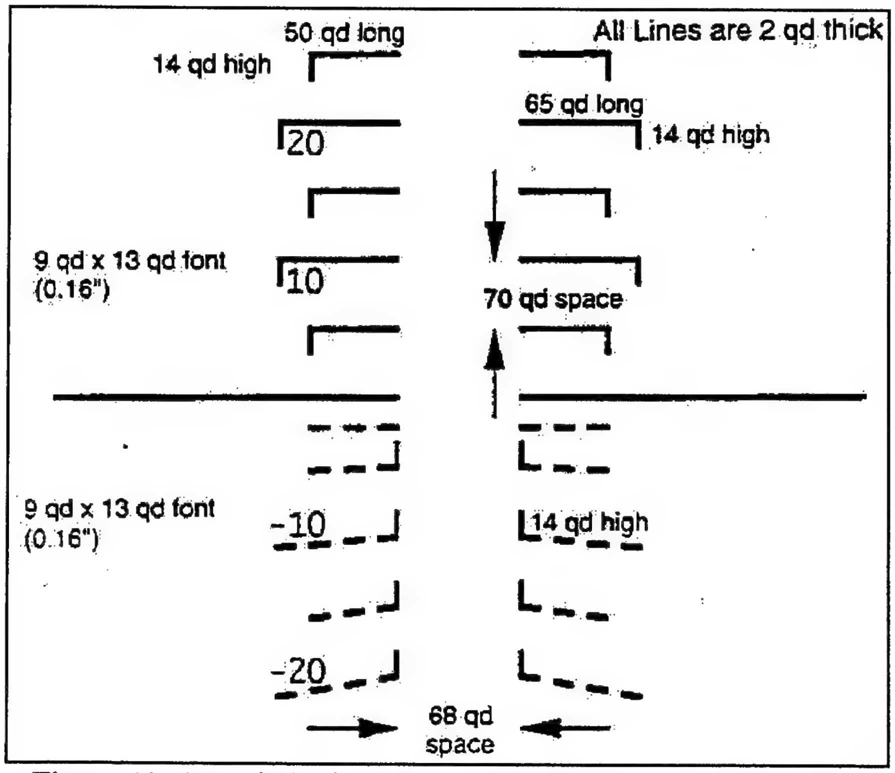


Figure 12. ADI Pitch Line Graphical Description [4:Appendix A]

3.2 Designing and Building Displays.

Designing and building the displays is fairly straightforward. Initially, all of the development toolsets have some sort of learning curve. This learning curve is influenced by user experience with cockpit displays, software programming experience, or general software background. Through the course of the research, the learning curve was quantified to determine the learning curve for each development toolset. Since two displays were designed with each development toolset, it makes sense that the second display will be somewhat easier to accomplish since the user has some experience with the tool and is more familiar with its capabilities. Nonetheless, the development times were measured for both displays in each development environment to analyze objective timing characteristics. When developing the displays using the three development tools, the same set of standards was followed for each display as discussed in Section 3.1. This ensured that each display had the same starting point and required functionality.

3.3 Immersing Displays in Simulation Environment.

After the displays are developed, they are embedded in a simulation environment to ensure that the initial display requirements and standards are met. In some cases, this phase can be time consuming. For example, one particular development tool may not be able to meet a requirement or standard for a display, such as an update rate requirement. Trade-offs must be made in these cases. The display designer can opt to use another development tool, which may incur high penalties in both time and money, or the requirement in question can be changed, modified, or simply ignored. For this research effort, no trade-offs were required as each development toolset was able to meet or exceed display requirements and standards.

Embedding the displays into a real-time simulation environment requires a sound simulation environment. For this research effort, the Re-configurable Avionics Modeling and Simulation System (RAMSS) is used. RAMSS uses a core tool set to simulate particular aircraft flight models in real-time. Currently, SIMPAC contains the aerodynamic flight models for the C-130, F-15, and A-7 aircraft. The RAMSS simulation system runs in the Microsoft Windows NT environment using SIMPAC models written in C and C++. The SIMPAC models also provide simple subroutines for embedding outside applications into the RAMSS simulation environment. These subroutines allow an application to register variable names with the SIMPAC models. These variables are used to retrieve aircraft flight information necessary to update the cockpit displays.

Using the provided SIMPAC subroutines, the appropriate data is obtained. The data is then converted to the proper format required by the particular display. For example, consider the Attitude Director Indicator (ADI), which dynamically moves according to aircraft roll. The actual aircraft model in SIMPAC uses units of radians while the OpenGL developed ADI uses units of degrees. The aircraft roll must be converted from radians into degrees for the OpenGL ADI to operate correctly.

At this point, the display is fully integrated with the simulation and is tested. Display testing occurs in three formats. First, the displays are tested for proper look and feel. This means that the ADI and HIS developed should look similar to the ADI and HIS as depicted in the *F-22 Air Vehicle Cockpit Design Description Document*. The displays should be executed sufficiently to test the entire range of motion of the displays ensuring that they behave as expected.

Next, the displays are examined in detail ensuring that the display characteristics match the requirements outlined in the *F-22 Air Vehicle Cockpit Design Description Document*. This part of the testing checks colors, locations, and sizes of various objects within the display. Small items that may have been overlooked can be extremely costly at this stage in development. As an example, consider a display that is required to be sized at 640x480. All of the display standards are based on the pixel locations on the 640x480 display. If the display is developed using a 600x400 format, every object within the scene will be placed according to the 600x400 window, which is incorrect. The display would have to be redesigned in the correct 640x480 format, changing the location of all objects within the scene. It is simple things like display size that can be overlooked in display development costing significant delays and costs.

The final phase of testing ensures that the displays operate correctly. As discussed above, this entails converting data into the proper formats expected by the display. Furthermore, this part of the testing ensures that the dynamic movement of the display is correct. For example, the heading of the aircraft must be correctly represented on the HSI during flight. If the HSI rotates clockwise instead of counter-clockwise with increasing aircraft heading, a simple sign error has occurred. These types of tests ensure correct operation of the display. This final testing phase also ensures that real-time performance requirements are also met. Recall that Chapter 1 defined real-time performance as a minimum update rate of 16Hz with a desired update rate of 30Hz. The reason behind the minimum of 16Hz is quite simple; the human eye can detect anything less than the 16Hz refresh rate as a jitter or delay in the display [3:1]. Each display contains a small amount of code (5 lines in C++) that calculates the display update rate. This data is output to the screen of each display so that the update rate can be measured and recorded. Both the ADI and HSI must run in real-time. If the display cannot run in real-time, it is modified to do so. Modifications may include using display lists (an OpenGL and Display Editor feature) or removing antialiasing algorithms (a feature in all three development toolsets). Display modifications for this research effort were not required as the ADI and HSI developed using the three development tools exceeded both the minimum 16Hz requirement and the desired 30Hz update rate.

3.4 Analyzing Development Toolsets Using Evaluation Techniques.

Throughout the development lifecycle, each of the displays has information and data recorded about the toolset used for its development. The data obtained are results from evaluation techniques discussed in Chapter 5. These techniques evaluate both subjective and objective criteria of the development toolsets. In addition to evaluating the development tools, the displays themselves are also evaluated for performance characteristics. Since the development process is largely subjective in nature, it is important to quantify the subjective characteristics of the development process. A series of questionnaires with weighted answers are used to quantify subjective criteria for each development tool. These criteria include learning curve, readability, user-interface, and maintenance. In addition, objective data is gathered from each developed display such as refresh rate, executable size, and development time. All of the data collected is used in an in-depth analysis to determine the advantages, disadvantages, and performance characteristics of each development toolset.

3.5 Executing Displays Across Different Hardware Configurations.

In a second experiment, each display is executed on several hardware configurations. This allows further characterization of the development toolsets and the displays created using them. Table 1 below shows the three hardware configurations used to test each of the displays generated using OpenGL, VAPS, and Display Editor.

Each configuration has unique characteristics that will test the individual displays developed using OpenGL, VAPS, or Display Editor. For instance, the HP Kayak machine has an OpenGL accelerated graphics processor that enables applications (i.e. displays) using OpenGL graphics bindings to perform better. The hardware platforms also vary in memory and CPU processing power. One might assume that the HP Kayak

Table 1. Three Hardware Configurations for Display Execution

Configuration		Processor	Memory (RAM)	Graphics Processor	Graphics Memory	Cost (Year)
1	Micron	Pentium 200-MHz	64MB	ATI Rage Pro Turbo	8MB	$2400 (1997)
2	HP Server	Dual Pentium 333-MHz	256MB	ATI Rage Pro Turbo	8MB	$4500 (1998)
3	HP Kayak	Dual Pentium 500-MHz	400MB	HP FX-6 Video	18MB	$12500 (1999)

performs better than the other machine based on sheer processing power, which may be a correct assumption, but this is not always the key to good performance. If the graphics cards were to be switched out between the Kayak and the Micron it is possible that the Micron would outperform the HP Kayak running certain OpenGL-based applications. This is especially true since the ATI graphics cards on the Micron and HP Server do not have OpenGL acceleration hardware on board its graphics processor.

The update rates are taken for each display running in each configuration. The display update rate measurement is then used to characterize dependencies on certain hardware devices (i.e. amount of memory, processing power, type of graphics board, etc.). Chapter 5 goes into more detail about this experiment.

3.6 Summary

The results of the evaluation techniques discussed above and an analysis of these results are presented in Chapter 6 with conclusions following in Chapter 7. The analysis covers the advantages, disadvantages, and characteristics of each display and development toolset. This research effort centers on this analysis providing engineers with critical information concerning the different development toolsets, such as development lifecycle, advantages, disadvantages, characteristics, and most importantly, cost. It is not meant to determine which development environment is better, but instead, to help cockpit display engineers make better decisions concerning hardware and software purchases for their simulation facilities.

IV. Display Development Methodology

4.0 Introduction

Section 3.2 briefly touched on the development of the displays. This chapter goes into greater detail about the design and development of the displays, the Attitude Director Indicator (ADI) and the Horizontal Situation Indicator (HSI). The two displays are designed using a linear sequential design model. Although this design process is directed specifically at these two cockpit displays, it could be applied to any human-machine interface type display development or, even more generally, to any software development project. The linear sequential model is modeled after the conventional engineering cycle. In fact, according to Roger Pressman in his book, Software Engineering: A Practitioner's Approach, the linear sequential model is the most widely used paradigm for software engineering. There are five activities in the linear sequential model. The five design activities are described in Table 2.

In general practice, the development phases in Table 2 should be accomplished in the order presented. It is typical in most software development environments that certain phases be revisited to clarify problems and resolve issues discovered in later phases.

Table 2. Five Phases of Linear Sequential Model

Phase	Description
Requirements Analysis	Function, Behavior, Performance, and Interfacing
Design	Structure, Architecture, Interface, Algorithm
Code Generation	Convert design description into machine code.
Testing	Statements, Externals, Results
Maintenance	Changes, Upgrades, etc.

4.1 Requirements Analysis

In software engineering projects the requirements analysis phase determines the function, behavior, performance, and interfacing requirements for the software to be developed. With this in mind, the engineer (analyst) must be familiar with the information domain of the software project. For this research, the information domain is aircraft cockpit design and layout. When dealing specifically with Department of Defense weapons system, the requirements analysis phase can be extremely time consuming, as the typical production cycle on weapon systems is 8-15 years [4].

For cockpit displays, requirements analysis describes the low-level details including such items as the size of objects, size of fonts, object colors and overall display size. The requirements analysis phase focuses on two main classes of display characteristics, static and dynamic. The static display characteristics represent the "look" of the display. These characteristics include font type, font size(s), object colors, object sizes, object spacing, scene size, etc. The dynamic display characteristics represent the "function" of the display. These characteristics include animation requirements, re-configurable information, warning lights, etc. There are also certain objects and messages that must appear at critical times during flight. For example, on the ADI display, a -2.5° pitch line must appear when the aircraft is accomplishing a landing or approach [4:Appendix A]. As another example, warnings are required when the altitude or airspeed is low. These static and dynamic display characteristics comprise the requirements analysis phase of the display design process.

As discussed in Chapter 3, the development team at Lockheed-Martin and Boeing has already accomplished the requirements analysis for the displays in this research. This standards document, titled *F-22 Air Vehicle Cockpit Design Description Document –* 1187B, is dated July 1996 [4]. This document fully details the design, layout, and function of every display to be used in the F-22 Raptor cockpit. The main document describes the dynamic functionality of the cockpit displays using textual descriptions while several appendices describe the "look" of the displays using pixel-by-pixel descriptions of display symbology.

4.2 Design Phase

With the requirements document in hand, the design phase begins. This phase centers on three attributes of the software program: data structure, software architecture, and interface representations [11:31]. Since this effort uses three distinct development environments, there are three separate design phases, one for each development toolset.

The OpenGL graphics package allows the designer great flexibility in deciding on data structures, architecture, and interfacing. For example, OpenGL contains a data structure called a display list. This data structure allows for a large number of graphical object definitions to be stored in a list. They need only be defined once and can be executed multiple times. For instance, since the structure of the ADI pitch ladder never changes (only its location), a display list is used, increasing performance of the OpenGL-based ADI. The architecture used for the OpenGL displays is structural in nature in that they are self-contained in a single compilation unit. The main routine calls a display subroutine, which in turn calls functions that update the dynamic characteristics of the display. The display is packaged in a single structure encapsulating both data and the processing that manipulates the data. As far as interfacing for the OpenGL displays, the use of include-files and libraries enables a connection to the appropriate RAMSS simulation variables (see Section 3.3) allowing simulated aircraft data to be sent to the displays.

The VAPS development environment is windowed allowing the display designer to use the point-and-click graphical user interface (GUI) to create the display. While the display designer has control over the interface, the VAPS software controls the data structures, architecture, and interfaces. The VAPS interfaces for this research use the C++ programming language and a C++ data structure called a "struct". The struct is a record-style structure that contains large amounts of heterogeneous information in a single data structure. By using this structure, a one-to-one mapping with variables in the VAPS channel file is achieved. The VAPS interface file is small (~150 lines of code) with one main loop continually sending data through the VAPS channel file to the VAPS display. Like the OpenGL interface, VAPS uses necessary include-files and libraries to connect RAMSS simulation variables to corresponding variables in the VAPS channel file (through the record data structure).

The *Display Editor* development toolset combines the function of both OpenGL and VAPS. Using a point-and-click window, the display designer creates the display by drawing primitives much like the VAPS toolset. Once drawn, the generated code is modified to provide for dynamic motion, similar to the OpenGL toolset. In fact, the code generated by *Display Editor* uses OpenGL graphics functions and libraries making it fully OpenGL compatible. *Display Editor* uses the same data structures as OpenGL, making use of OpenGL display lists as well as C++ array and record structures. Display Editor uses an Object Oriented architecture encapsulating the data with the functions that manipulate the data. Finally, *Display Editor* uses the same interfacing schemes as VAPS and OpenGL, using the appropriate include-files and libraries to connect to the RAMSS simulation variables.

It is important to note at this point that the design phase does not involve any software coding. The above discussions reveal the decisions made in the design phase concerning the different data structures, architectures, and interfacing schemes used for this research effort. In practice, certain aspects of the design phase may be revisited for errors, future upgrades, or general maintenance. The following phase, the code generation phase, is concerned with converting the design decisions, in coordination with the requirements analysis document, into machine-readable code.

4.3 Code Generation Phase

The code-generation phase for the displays uses the requirements and standards document and results from the design phase. At this stage, OpenGL, VAPS, and *Display Editor* begin to show major differences.

In many software engineering ventures, the code-generation phase can be automated, provided the requirements analysis and design phases are performed in a detailed manner. For this research, both VAPS and *Display Editor* take advantage of automated code-generation. As discussed before, the VAPS toolset uses a point-and-click, windowed environment. Once the display is drawn, dynamics defined, and the channel file developed, the VAPS display can be sent through a code generator. This code generator produces a C++ executable application. As seen in the design phase discussion above, a separate interface application is also required in order for an outside application (the RAMSS SIMPAC models) to communicate with the VAPS display. This interface, written in C++, uses the necessary include-files and data structures decided upon in the design phase.

Display Editor also uses automated code-generation techniques. The contract under which *Display Editor* is being developed demands OpenGL compliance. With this in mind, SCS Engineering developed a toolset that uses a code generator to generate OpenGL code using C++ bindings. Using a similar point-and-click windowed environment the display is drawn and the code generated from the drawing. The generated code contains no dynamic OpenGL definitions, only primitive OpenGL object definitions. The display designer must then add the dynamics and interfacing schemes to the generated code. A slight difference between *Display Editor* and VAPS is that the interface and dynamic definitions can be integrated with the generated code without building a second interface application.

The OpenGL toolset has perhaps the simplest yet most time consuming form of code generation. Displays developed using OpenGL are hand-coded. This research uses the OpenGL bindings for C++ running in a Microsoft Windows NT environment. Armed with the requirements analysis document and design phase decisions, the display designer hand-codes the display, creating the primitive objects, dynamic definition, and interface all in a single phase of code generation. Building an application by hand can be time consuming, and even more so given the requirement of OpenGL bindings and the detail of cockpit displays.

4.4 Testing Phase

Once the code has been generated, the testing phase begins. This portion of the linear sequential model focuses on a line-by-line test of each coded program. In addition, this phase tests the basic functionality of each program ensuring that defined input produces results that are expected. For each of the displays, a line-by-line test would be extremely time consuming, especially in terms of the volume of the generated code for the displays developed in VAPS and Display Editor. Instead, a test method is developed for each display that sends test data to the display to exercise the code, while at the same time testing the display's functionality.

Since both the OpenGL and the *Display Editor* displays use OpenGL functions, the test routines are also based on OpenGL. For example, OpenGL provides a keyboard function that allows a key on the keyboard to be defined for some functionality. For the OpenGL based displays the keyboard function used the keys on the number pad to define such things as pitch and roll for the Attitude Director Indicator (ADI) displays. In addition, the OpenGL mouse function was used to define a pop-up menu to allow for switching between modes for the displays. On the ADI displays, the pop-up menu allows the display to be switched from day to night mode, changing the colors used on the displays. In this manner, the OpenGL and *Display Editor* displays were tested for functionality as well as program correctness.

Testing for the VAPS displays is simpler than for the OpenGL and *Display Editor* displays. Since the interface application simply takes input data from the RAMSS simulation, converts the data, and then sends it on to the display, a test routine is defined that replaces the simulation. The test routine simply increased and decreased values for the variables to be sent to the displays to test the full range of values. Running the interface with the test routine and the VAPS display allows the interface and display to be tested for functionality and correctness.

4.5 Maintenance Phase

Software will inevitably go through changes following delivery to the user. If errors are found, the functionality of the software must be changed to adapt to some new environment, or the user may want to upgrade hardware or software to boost performance. The maintenance phase is a continual execution of the previous phases of the linear sequential model. A change or upgrade must go through requirements analysis, design, code-generation, and testing, although it is typically small in scale compared to a completely new software application.

For this research effort, the maintenance phase can only be assessed within the limited amount of time provided for research. The evaluation techniques (discussed in Chapter 5) reveal how the maintainability of the software can be quantified and evaluated to a limited degree. For the OpenGL and *Display Editor* displays, changes or upgrades for maintenance reasons are typically more time consuming, as the changes must be hand-coded. With the VAPS displays, any upgrades or changes can simply be made in the point-and-click window environment, taking significantly less time. The VAPS changes, however, must also be reflected in the channel file and the interface application.

4.6 The Displays

Six crewstation displays were designed and developed following the Linear Design Model outlined above. An Attitude Director Indicator (ADI) and a Horizontal Situation Indicator (HSI) were designed using OpenGL, VAPS, and *Display Editor*. As discussed above in the requirements analysis, the same F-22 documentation and military standard, *F-22 Air Vehicle Cockpit Design Description Document* – 1187B, guided the design assuring a consistent design and display format.

The OpenGL displays were hand-coded. Functions provided by the OpenGL graphics libraries and the GL Utility Toolkit (GLUT Version 1.2) were used to create the proper display formats and dynamic animation. All necessary files for the OpenGL displays were obtained through the Silicon Graphics website (http://www.sgi.com). The OpenGL developed ADI and HSI are shown below in Figures 13 and 14.

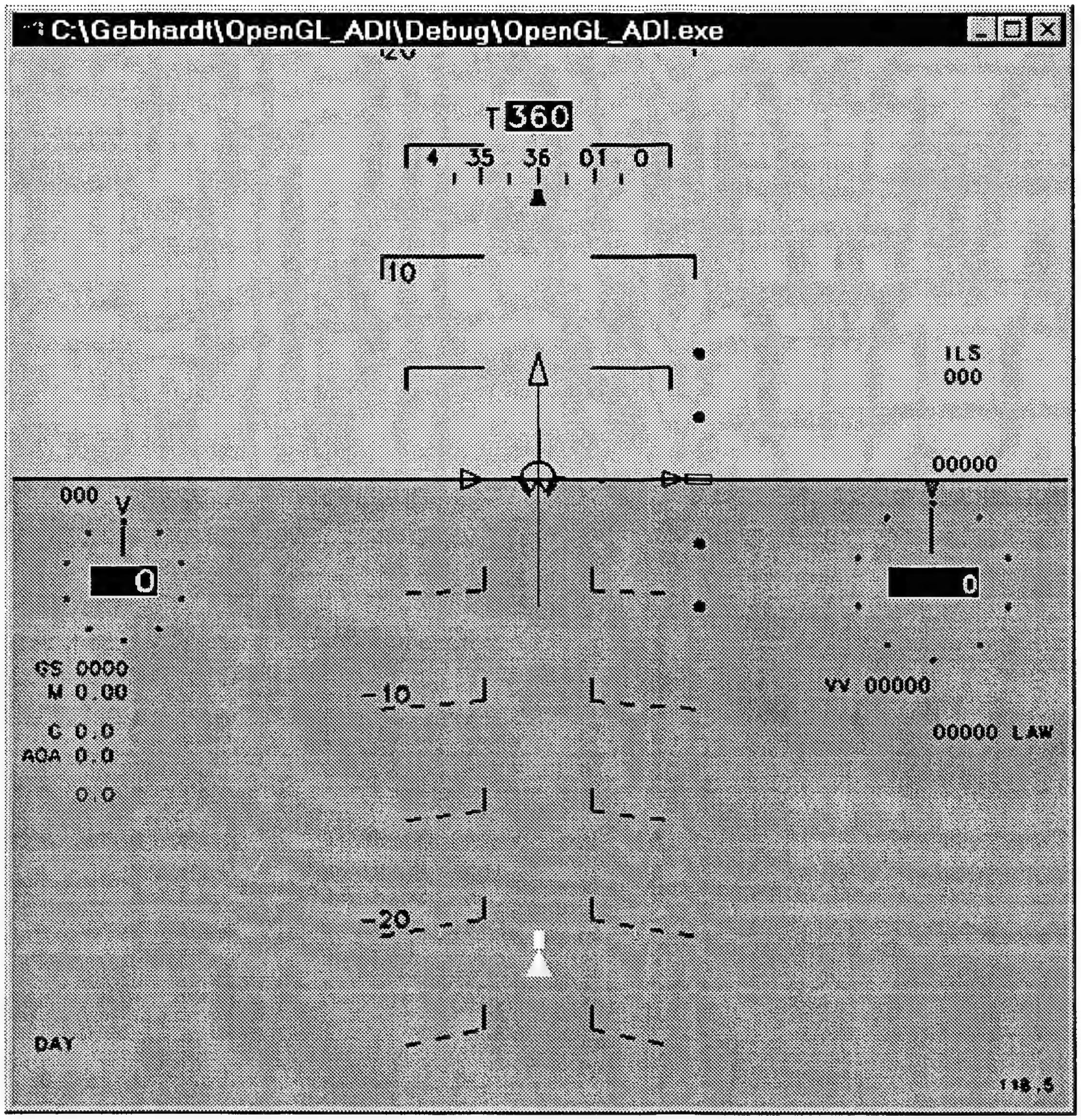


Figure 13. OpenGL hand-coded Attitude Director Indicator

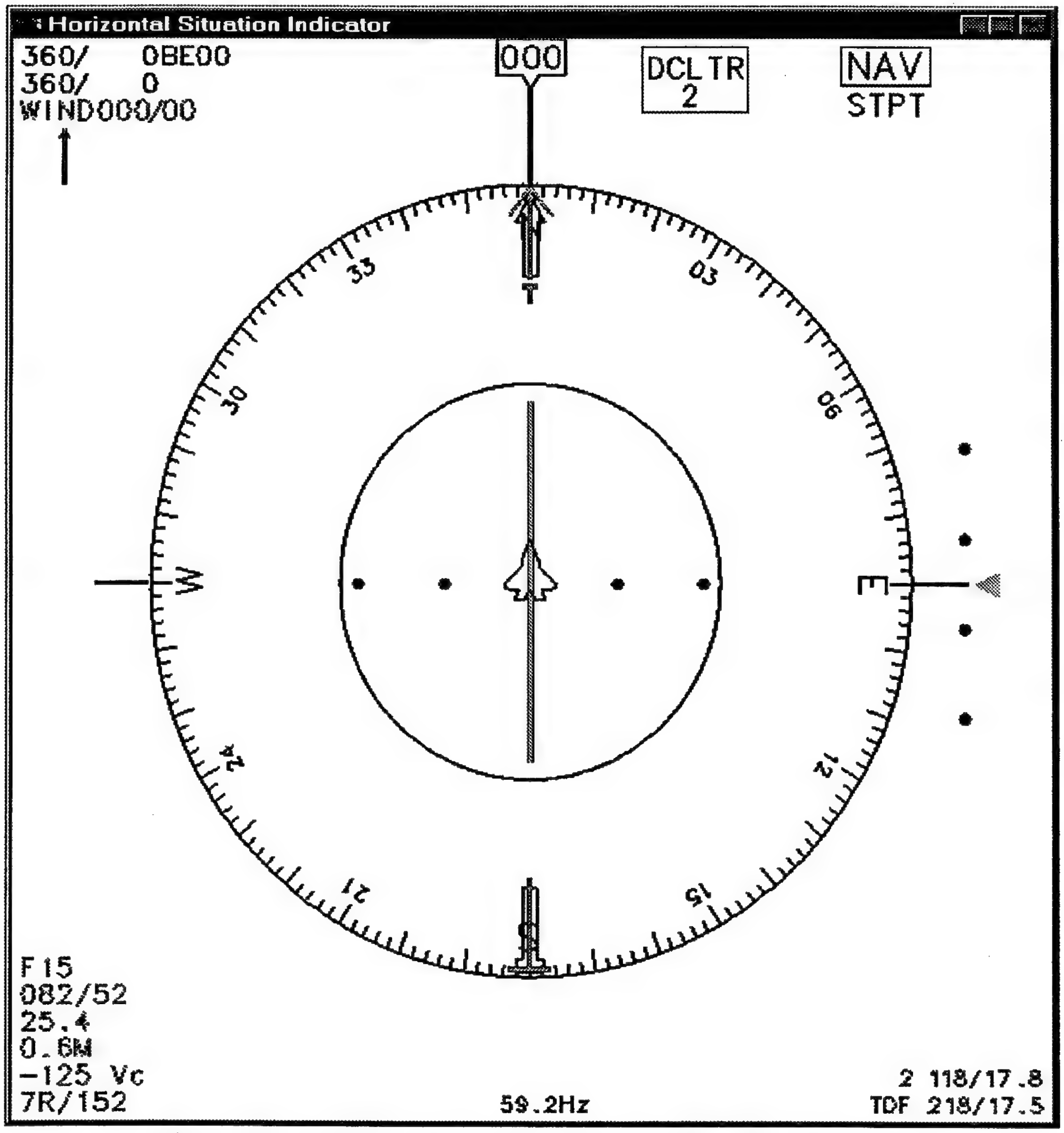


Figure 14. OpenGL hand-coded Horizontal Situation Indicator

The OpenGL and GLUT libraries provided all necessary functionality to meet the display requirements established by the F-22 Development team at Lockheed and Boeing. Although the Advanced Architecture and Integration Branch's simulation system does not currently use all of the display objects and capabilities, the necessary functionality is in place for future crewstation integration.

The VAPS software licenses (version 5.1) were obtained through Virtual Prototypes, Inc. (VPI) with all the necessary functionality included in the package. Again using the Linear Sequential Model, the two displays were developed. Seen in Figures 15 and 16, the ADI and HSI look very similar to the displays built using OpenGL.

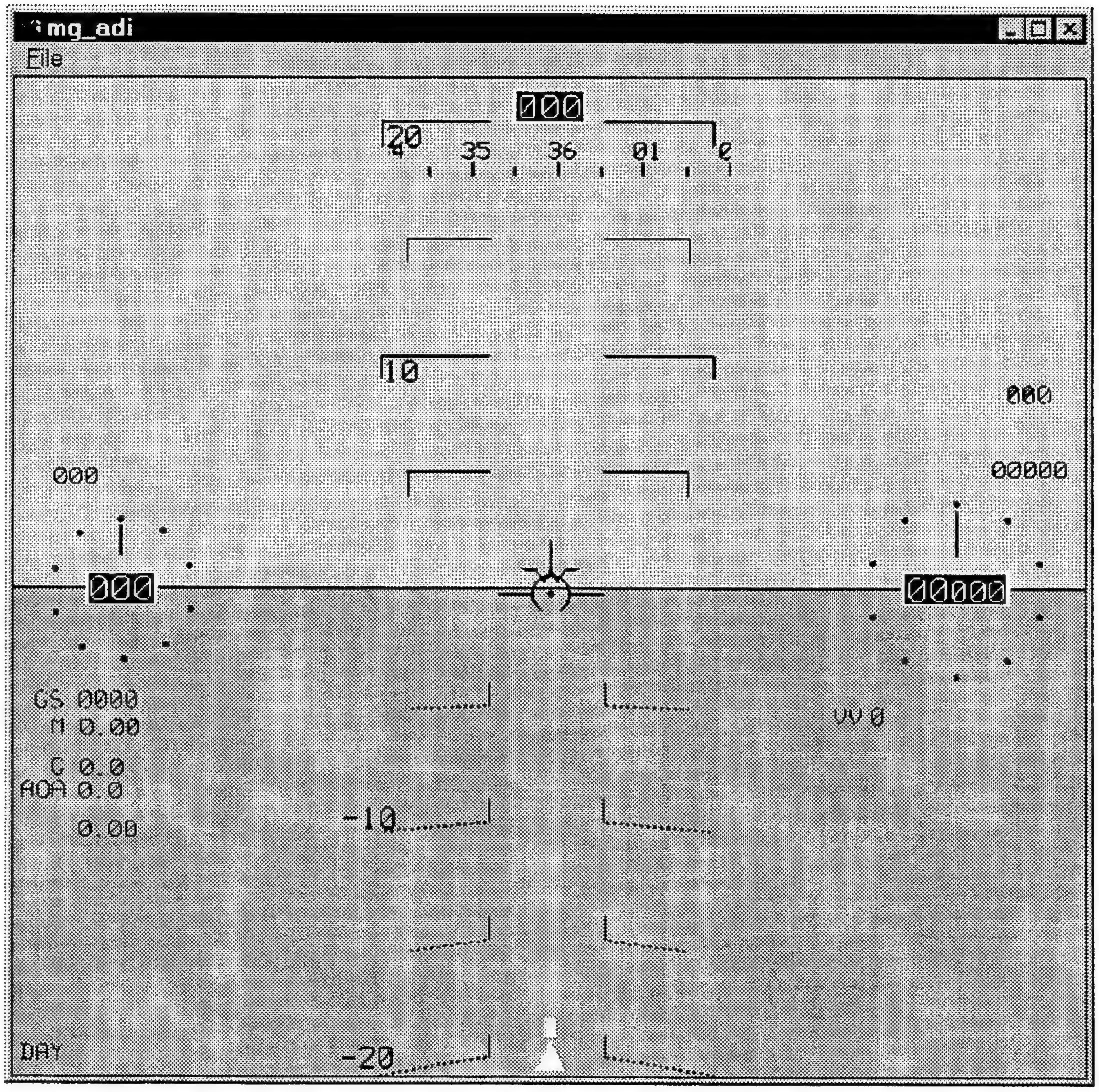


Figure 15. VAPS developed Attitude Director Indicator

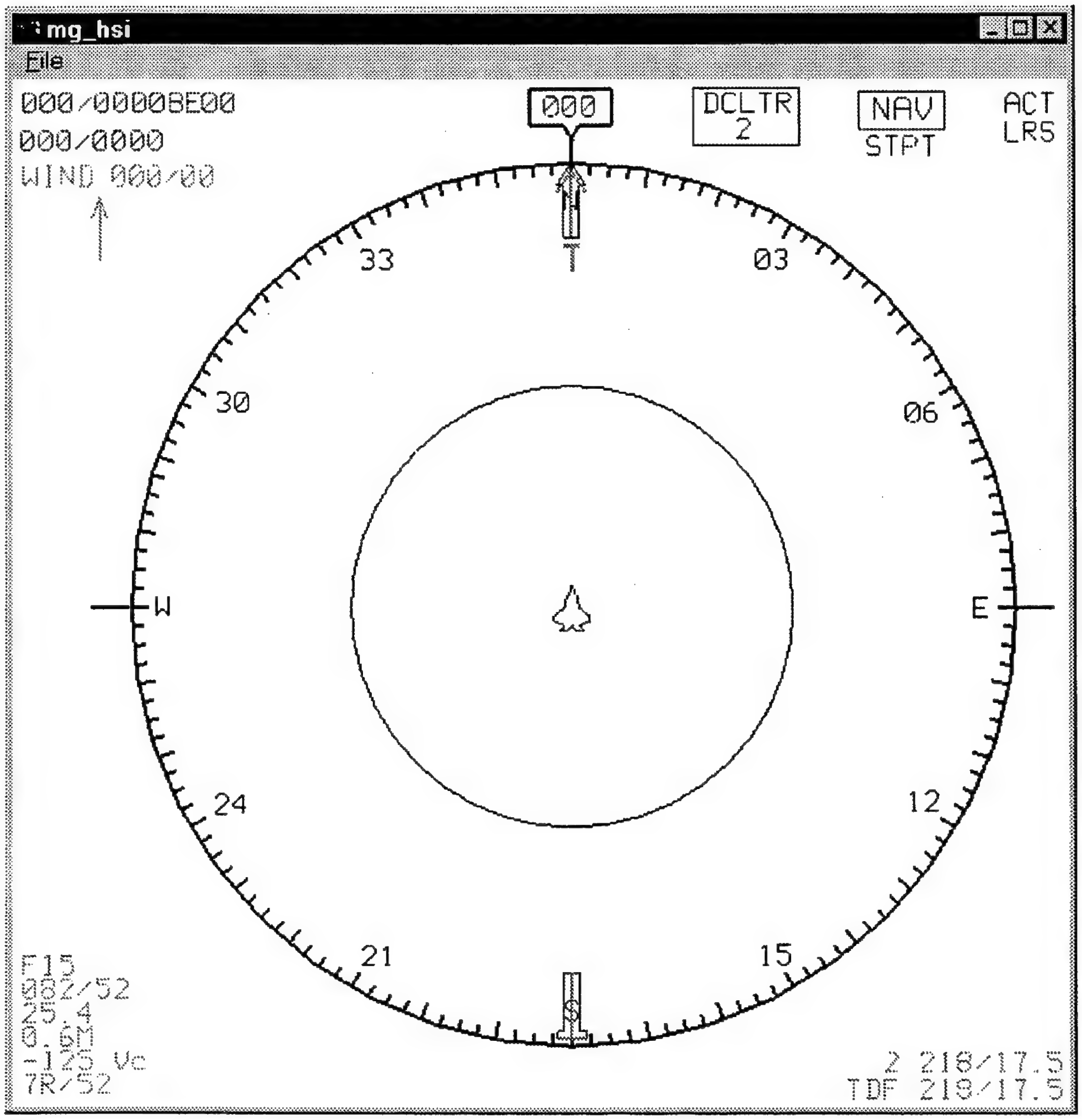


Figure 16. VAPS developed Horizontal Situation Indicator

The most significant issue to come out of the VAPS development process dealt with creation of display executables. VAPS companion package, the C-Code Generator (CCG), required numerous phone calls to the technical support staff at VPI to correct minor undocumented problems with configuration and support file requirements. As with the OpenGL displays, all functionality is provided by the displays even though some of the objects and capabilities will not be used in the Advanced Architecture and Integration Branch's simulation system.

Finally, the Display Editor software licenses were provided through a Small Business Innovative Research (SBIR) contract between SCS Engineering, Inc and the Advanced Architecture and Integration Branch. In its present state, the Display Editor toolset requires both a graphical automatic code generation interface and manual software programmer interface to develop the displays. These displays were developed through the Advanced Architecture and Integration Branch's SBIR contract with SCS Engineering, Inc. The staff at SCS Engineering, Inc., while not participating in the actual drawing of the displays, played an important part in dynamic display animation. For example, due to its immaturity, the Display Editor tool did not have the functionality required to define the Attitude Director pitch ladder. The solution was to send the drawn ADI display to the staff at SCS and have them engineer the pitch ladder manually using OpenGL. The resulting development times discussed later in Chapter 5 reveal this immaturity problem. The Display Editor displays are shown in Figures 17 and 18.

The Display Editor tool is in its early stages of development and this research revealed some problems that will be corrected with future versions. However, the current version contains enough functionality to build the required capabilities for the two displays.

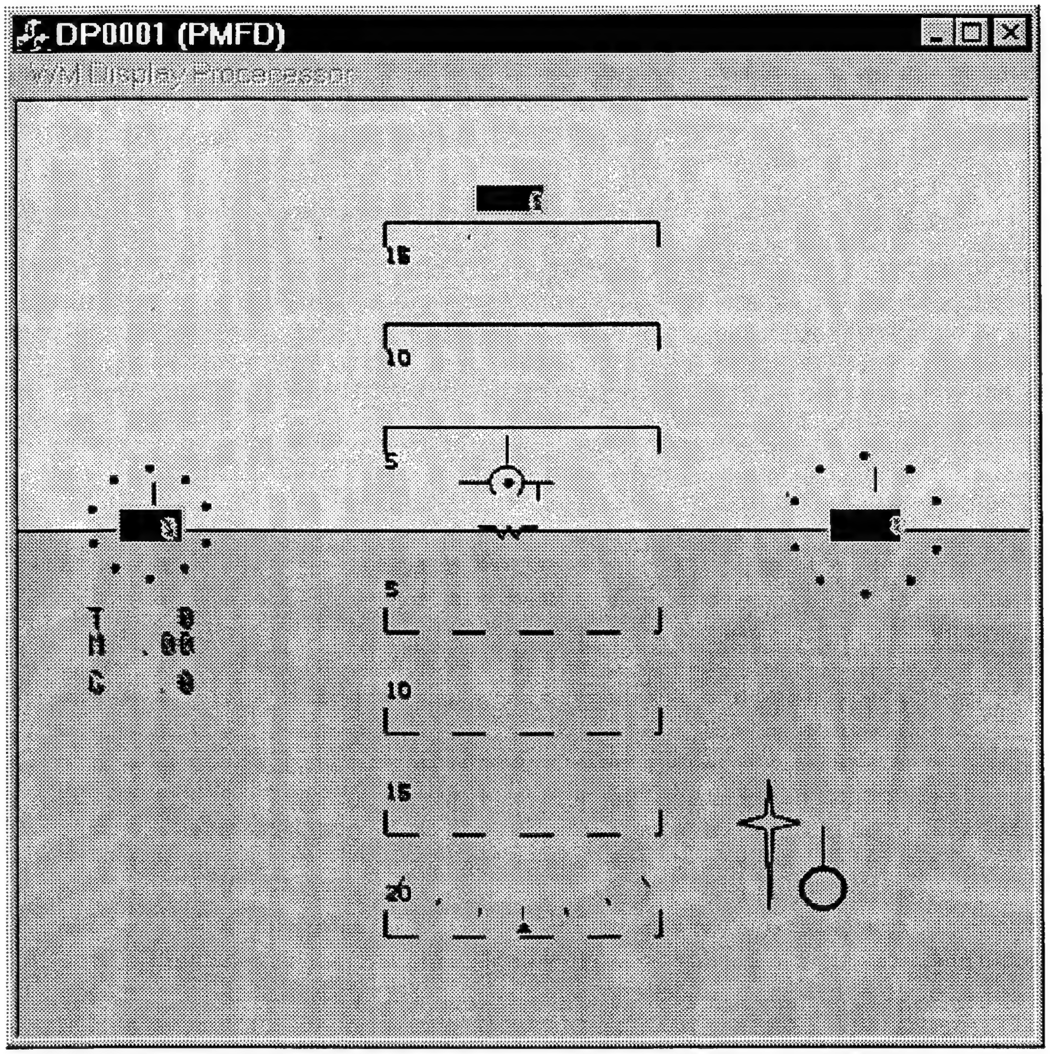


Figure 17. Display Editor developed Attitude Director Indicator

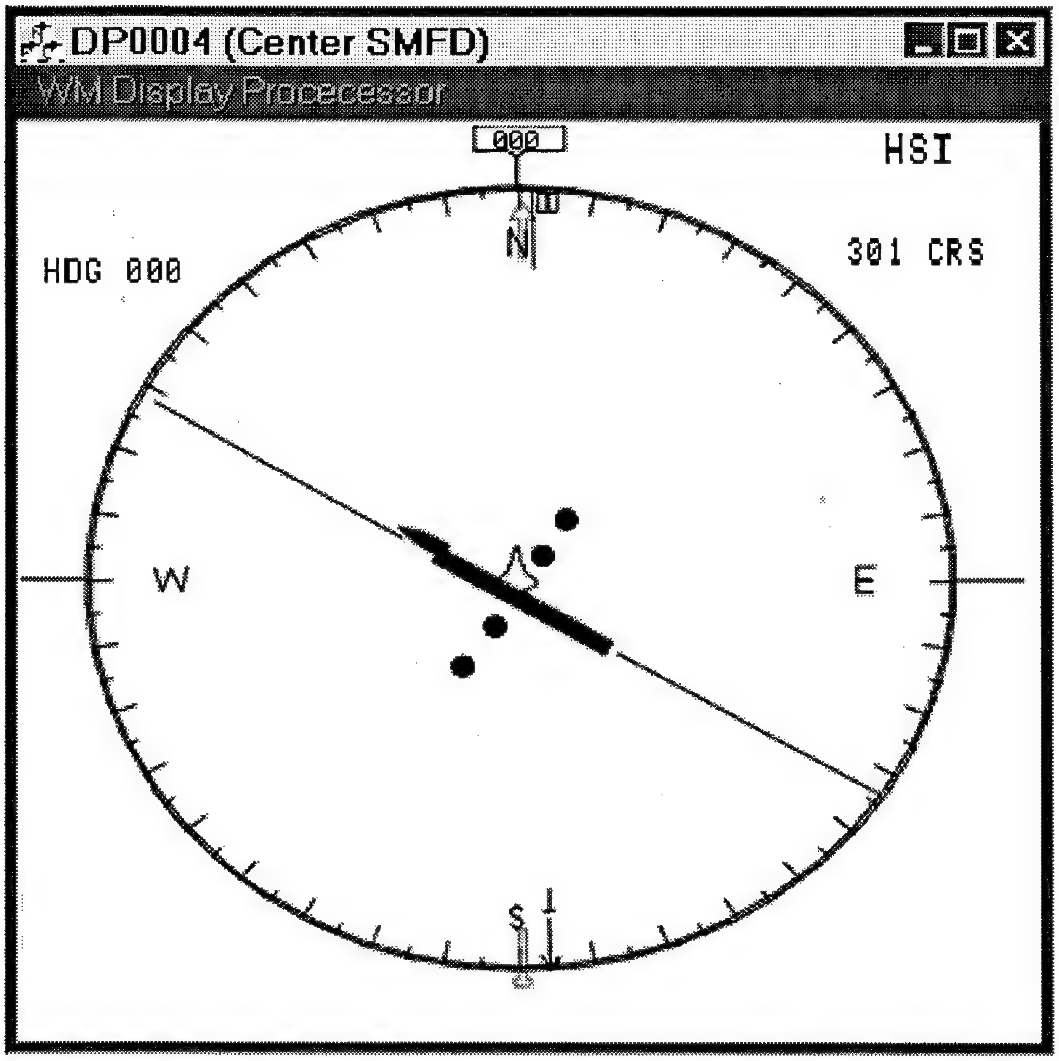


Figure 18. Display Editor developed Horizontal Situation Indicator

V. Display Development Evaluation Techniques

5.0 Introduction to the Two Experiments

This research effort focuses on an in-depth comparative analysis of the three cockpit display development tools; OpenGL, VAPS, and Display Editor. In order to accomplish a thorough analysis, two experiments are accomplished. These experiments were briefly introduced in Chapter 3. This chapter goes into detail about the two experiments and how they are conducted.

The two experiments form a foundation for the analysis of the three development tools. The first experiment examines both subjective and objective criteria for the cockpit display development lifecycle, while the second focuses on the performance characteristics of the developed displays. As discussed in Chapter 3 and Chapter 4, every cockpit display goes through an initial requirement phase establishing the dynamic definition and characteristics of the display. The two experiments examine how VAPS, Display Editor, and OpenGL implement these characteristics as well as how the displays perform.

Chapter 4 discussed the development and design of the displays using the three development toolsets. The first experiment gathers data throughout the display development process. The goal of this experiment is to gather both subjective and objective data for use in the analysis of the three development tools. Understandably, subjective criteria cannot be easily measured or quantified. In order to quantify certain subjective criteria about each development toolset, a set of questionnaires is generated. The questions within each questionnaire contain weighted answers about key subjective criteria such as learning curve, user interface, extensibility, portability, and readability. Once complete, each question within the questionnaire is answered and given an appropriate weight quantifying a particular subjective criterion for the development toolset. Objective data is also collected for the first experiment to include criteria such as tool cost, man-hours for development, executable size, update rates, CPU usage, and memory usage. Together, the subjective and objective data gathered for each toolset form a foundation for analyzing the effectiveness of the display development toolset as discussed in Chapter 6.

The second experiment examines the displays after the development lifecycle is complete. Data gathered for this second experiment is objective in nature. The cockpit displays are executed on the three hardware configurations of Table 1 in Chapter 3. The display update rates for each cockpit display, running in each configuration, are measured and recorded. An analysis of the data, accomplished in Chapter 6, shows the hardware performance characteristics and dependencies of the development toolsets.

The analysis of the data gathered in these two experiments reveals advantages, disadvantages, and performance characteristics of the three development toolsets. Display engineers can perform cost-benefit analyses for any cockpit display development software using the questionnaires developed for the first experiment. The analysis and results also provide key information about hardware configurations that best suit each of the development tools. The analysis reveals the advantages, disadvantages, and performance characteristics of the development toolsets. An engineer may accept certain disadvantages of a development toolset in order to capitalize on specific advantages.

5.1 Display Development Experiment

As discussed previously, data gathered from this experiment is both subjective and objective in nature. Subjective data is gathered in the form of weighted answers to several questionnaires about key subjective criteria pertaining to the development lifecycle for each toolset. The objective data is simply measured and recorded. This experiment examines the effectiveness of each development toolset in meeting the requirements set forth in the F-22 standards document.

Subjective criteria like tool learning curve, user interface, readability, extensibility, and maintainability are examined throughout display development. The objective criteria examined include development time, executable size, CPU usage, and memory usage. Learning curve and user interface are primarily user requirements while other subjective criteria are display requirements.

The questions developed for each individual subjective criterion questionnaire come from several different sources. These sources include personal experience and the personnel at the Advanced Architecture and Integration Branch and SCS Engineering. The personnel at the Advanced Architecture and Integration are highly educated and experienced in matters dealing with aircraft development, especially in the department of aircraft and cockpit simulation. Lastly, sources such as Roger Pressman's book, Software Engineering: A Practitioner's Approach, and its detailed discussions about function points, provide an excellent source for criteria-based questions.

The data gathered from these subjective questionnaires and objective measurements are used in the analysis to determine the advantages and disadvantages of the OpenGL, VAPS, and Display Editor development environments. The results and analysis of the data gathering are presented in Chapter 6, while conclusions from the analysis follow in Chapter 7. The following sections go into greater detail about the subjective and objective criteria and how each will be tested, quantified, or recorded. Included with each criterion explanation is a table of the quantifying questions to be asked about each development tool.

5.1.1 Development Tool Learning Curve

The learning curve of a software development tool can be one of the most significant parts of a decision to use a particular tool. In fact, quantifying the learning curve can be extremely difficult because it has many influences such as level of programming experience, general code knowledge, general software background, online help, and availability of examples. This is a difficult subjective criterion to quantify, but also one of the most important. For this research, quantifying the learning curve is accomplished through a series of questions about the influences on the learning curve. Table 3 shows the 12 questions chosen to quantify the learning curve for each development toolset.

Table 3. Learning Curve Questionnaire

#	Question	Weighted Answers (0=worst 5=best)
1	Is a user's manual/documentation available?	**0 1 2 3 4 5**
2	Is the documentation understandable?	**0 1 2 3 4 5**
3	Are examples included in software deliverable?	**0 1 2 3 4 5**
4	Are examples available elsewhere (i.e. web)?	**0 1 2 3 4 5**
5	Are examples complete and well documented?	**0 1 2 3 4 5**
6	Are courses available for the toolset?	**0 1 2 3 4 5**
7	Does analyst/designer have display experience?	**0 1 2 3 4 5**
8	Does analysts/designer have coding experience?	**0 1 2 3 4 5**
9	Is knowledge of coding NOT needed?	**0 1 2 3 4 5**
10	Do Industry leaders use the development tool?	**0 1 2 3 4 5**
11	Is the software specialized or general?	**0 1 2 3 4 5**
12	Is help available online?	**0 1 2 3 4 5**
13	Is help available through software producer?	**0 1 2 3 4 5**
0 = Worst 5 = Best	Score :	/65

These questions about such learning curve influences as experience, background, documentation/users manual, courses available, examples available, tool specialization level, and use in industry, are asked to quantify the influences on the learning curve for each development tool. By asking questions about the influences on learning, one can analyze the learning curve itself. One significant influence that is not included in Table 3 is the user interface. The user interface is such an integral part of today's software development tools that it has its own separate category for this experiment.

Documentation included with a software tool often has a great influence on the learning curve. While some software development tools provide a hardcopy of a user's manual, others provide the user's manual online. The ability to read and understand that user's manual is important, especially if some feature of the tool is difficult to understand or implement without it. In addition to the user's manual, another significant issue is the availability of examples. Some software packages include examples with the software deliverable, while others provide examples elsewhere, like the World Wide Web or through additional software purchases. The extent of these examples contributes to ones ability to learn the tool. OpenGL, for instance, has countless examples located throughout the Internet. This is a testament, not only to OpenGL's broad use, but its breadth of applicability. Finally, technical support and staff knowledge of common problems or mistakes can make or break the effectiveness of the development tool.

5.1.2 Development Tool User Interface

Typically in today's software environment, the user interface is the most important part of a software tool, playing a significant role in the learning curve. For this reason, the user interface is a separate subjective criterion for this experiment. The questions in this category (Table 4) are concerned with the extent of the user interface, understandability, and the documentation included about the interface. One primary problem with software tools is that they have very well thought-out user interfaces, but they fail to include the appropriate documentation to help the user understand the functionality of the interface.

A software tool with a graphical user interface (GUI) can have great benefits to a user. The tool can be extremely powerful while making it fairly simple for someone to use. In the cockpit display environment, the graphical drawing of the objects for the display is extremely helpful. Being able to draw a circle, square, triangle, etc. and even

Table 4. User Interface Questionnaire

#	Question	Weighted Answers (0=worst 5=best)
1	Is the user interface graphical (GUI)?	**0 1 2 3 4 5**
2	Does tool contain standard graphical shapes (i.e. circles, squares, etc.)?	**0 1 2 3 4 5**
3	Are tool bars for color, size, shape, etc. available?	**0 1 2 3 4 5**
4	Is there an object library (i.e. dials, tapes, etc.)?	**0 1 2 3 4 5**
5	Can projects be saved in graphical format?	**0 1 2 3 4 5**
6	Is there an online help engine?	**0 1 2 3 4 5**
7	Does the help engine provide examples?	**0 1 2 3 4 5**
8	Is the help engine detailed?	**0 1 2 3 4 5**
0 = Worst 5 = Best	Score :	/40

color the objects using a point-and-click environment can lower the display development time significantly.

Furthermore, the availability of an object library within the tool (library of defined objects like dials, switches or lights, for example) can also help the user in developing cockpit displays. Finally, the display designer benefits from the availability of a help engine when problems are encountered or they don't quite understand a feature of the interface or how to take advantage of it.

5.1.3 Display Readability

Readability is an essential requirement for cockpit displays. The pilot must be able to quickly and easily read the information presented on the display. The choice of font(s), font size(s), and the width and length of lines can be critical to readability. Since most displays are dynamic in nature, the fonts, lines, objects, and entire scenes will rotate and translate according to the dynamic definitions of the given display. For this reason, the F-22 cockpit displays require the use of antialiasing for lines, polygons, and fonts.

Antialiasing can greatly reduce the jagged edges seen in a graphical scene. Figure 19 shows an example of antialiased lines and lines without antialiasing. The Attitude Director Indicator, for example, contains a number of lines for the pitch ladder and as the aircraft rolls, antialiasing smoothes the jagged edges of the pitch lines, making the display more readable. Most graphics packages offer antialiasing algorithms that reduce the effects of aliasing as seen above. OpenGL, VAPS, and Display Editor each have various options for antialiasing. Objects that do not use antialiasing algorithms cause unwanted distortion of fonts and lines that can lead to pilot error and eye fatigue.

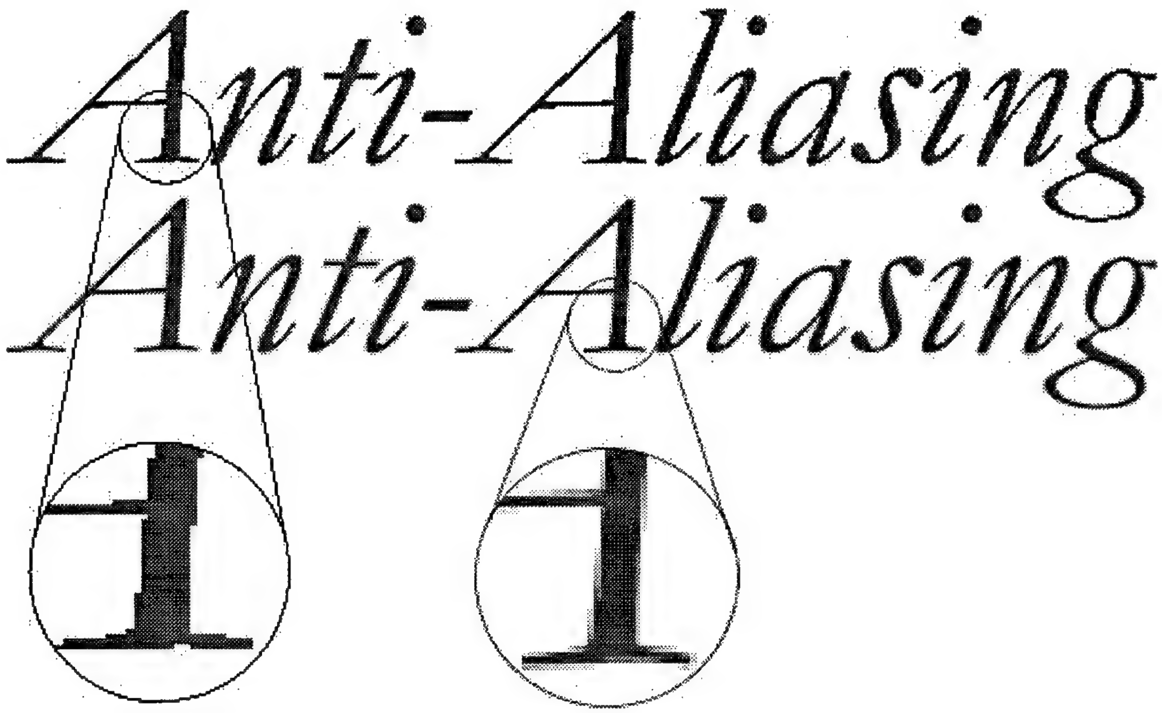


Figure 19. Antialiasing Example

Antialiasing, while an important requirement, is not the only measure of readability. The number of fonts available, with the ability to scale them, is another contributing factor. Selecting appropriate font sizes, line widths, and line lengths contributes to the readability of a display. Once completed, the displays are examined for general font and object clarity to answer questions 9, 10, and 11 in Table 5.

Table 5 does not reflect some other influences on display readability. Things such as color and scale contribute, but these items will have already been decided in the requirements analysis phase of display design. The development environment influences readability by enabling object rotation and scaling. Answers to the above questions allow for the readability of the displays developed with each development tool to be quantified.

Table 5. Display Readability Questionnaire

#	Question	Weighted Answers (0=worst 5=best)
1	Is font anti-aliasing a tool option?	**0 1 2 3 4 5**
2	Is line anti-aliasing a tool option?	**0 1 2 3 4 5**
3	Is polygon anti-aliasing a tool option?	**0 1 2 3 4 5**
4	Is implementing anti-aliasing easy?	**0 1 2 3 4 5**
5	Does the tool have a large # of fonts (i.e. > 15)?	**0 1 2 3 4 5**
6	Can lines, polygons, and objects be dynamically resized (i.e. changing the properties of them)?	**0 1 2 3 4 5**
7	Are the fonts scalable (i.e. do they change with window resizing)?	**0 1 2 3 4 5**
8	Are they vector fonts (i.e. do they rotate with the display)?	**0 1 2 3 4 5**
9	Are display fonts clear and readable?	**0 1 2 3 4 5**
10	Are display objects clear (i.e. distinct edges)?	**0 1 2 3 4 5**
11	Are display objects readable (i.e. visually pleasing)?	**0 1 2 3 4 5**
	0 = Worst 5 = Best Score :	/55

5.1.4 Display Portability

As a requirement for a simulation environment, developed cockpit displays must be portable. This means that the software-coded display executable can be moved from one hardware platform to another without major changes to the software. It is preferable that the displays have the ability to be ported to the new platform without the need for the display to be recompiled. OpenGL, VAPS, and Display Editor all have the ability to generate an executable (.exe file) for a particular display. The requirements for creating the executable, packaging it with any needed supplementary files, and running the display on another platform all contribute to the display's portability. The questionnaire found in Table 6 contains questions about the portability of the coded displays.

Table 6 shows that the portability measure for a display is influenced by a number of factors. Recompiling an executable for the target is time consuming and expensive, as compiler licenses are required. In addition, licensing issues for the development tool itself can also be a factor influencing display portability.

Table 6. Display Portability Questionnaire

#	Question	Weighted Answers
1	Can the display be built as a single executable?	**0 1 2 3 4 5**
2	Is executable generation simple and straightforward?	**0 1 2 3 4 5**
3	Are a small number of support files required (< 5)?	**0 1 2 3 4 5**
4	Can the display be ported without recompilation?	**0 1 2 3 4 5**
5	If recompiling required, are code changes simple and straightforward (if not, enter 5)?	**0 1 2 3 4 5**
6	Are licensing issues avoided for portability?	**0 1 2 3 4 5**
	0 = Worst 5 = Best Score :	/30

5.1.5 Display and Development Tool Extensibility

In addition to being portable, the software coded display needs to be extensible as well. For this research, extensibility is defined as a measure of the development tool's ability to allow the programmer to customize, reuse objects, add new functions, and modify the behavior of a developed display. Once a display is completed, features may need to be added or changed. An extensible development environment allows for these features to be added without difficulty. In addition, some objects within a display scene have the potential to be reused in other displays. For example, the compass wheel on the HSI may be reused as the heading indicator on a TACAN Display. The modularity and reusability of the objects within the display contributes to the extensibility of the display and development toolset. Table 7 shows the questions quantifying tool extensibility.

Table 7. Display Extensibility Questionnaire

#	Question	Weighted Answers (0=worst 5=best)
1	Can extensions and functions be added to the display at the graphical level (i.e. point-and-click)?	**0 1 2 3 4 5**
2	Is object-oriented programming used (modularity)?	**0 1 2 3 4 5**
3	Is display modular (i.e. groupable and selectable objects that can be copied)?	**0 1 2 3 4 5**
4	Does the display contain sufficient comments?	**0 1 2 3 4 5**
5	Do the comments have meaning?	**0 1 2 3 4 5**
6	Can objects in the display be reused?	**0 1 2 3 4 5**
7	Is reuse of objects simple (i.e. cut and paste)?	**0 1 2 3 4 5**
8	Is adapting the reused object simple and straightforward (i.e. modifying size, color, etc.)?	**0 1 2 3 4 5**
	0 = Worst 5 = Best Score :	/40

Modification and extension using a graphical user interface (i.e. point-and-click) is typically simple, straightforward, fast, and inexpensive. Development tools that have a graphical user interface do not require code manipulation and programming knowledge and are highly desired. In addition, object-oriented programming is a practice used in many software development environments today. The practice of object-oriented programming allows for easy code reuse because object-oriented code is modular by definition. Meaningful comments also contribute to extensibility. Comments help readers understand the functionality of the code and whether a particular piece of code can be reused for other functions. Comments that lack meaning contribute little, other than code separation.

5.1.6 Display Maintenance

Maintainability is the ease with which a display can be corrected if an error is found, adapted if the environment changes, or enhanced if a new feature is required. There is no way to measure maintainability directly, so a simple time-oriented metric is used. The mean-time-to-change (MTTC) is the time it takes to analyze a change, design the modification, implement the change, test it, and then integrate it [11:93]. On average, displays developed using toolsets that have maintainable code have a lower MTTC than those developed with tool sets that do not have maintainable code. The MTTC rating is used to quantify the maintainability of the software-coded displays generated from OpenGL, VAPS, and *Display Editor*.

For this research, the quantification of maintainability is limited to the overall time period for the research. However, once the displays were developed, some changes were required. Objects in the wrong location, incorrect sizes of fonts, and a required feature not implemented are all types of changes that needed to be made. Using these necessary changes, the MTTC was measured and recorded. For the OpenGL displays, several measures were taken as more semantic errors were found. This is logical since the OpenGL displays were hand-coded. Chapter 5 presents the errors encountered and the MTTC rating for each development toolset.

5.1.7 Display Update Rate

Maintaining real-time performance of a display is important, especially in the simulation environment where aircraft models are running. Update rates for most dynamic cockpit displays, like the ADI and HSI, must maintain real-time performance. Referring back to the Chapter 1 definition of real-time performance, displays must update at a minimum of 16Hz with a desired update rate of 30Hz. Critical aircraft information must be available to the pilot as fast as possible. There are some displays, however, that do not require real-time performance, such as a general status page reflecting fuel load, weapons load, and waypoint information. It is important to note that the display update rate will change as the CPU executes more programs. Running the display by itself allows for the maximum performance of the display. The most important update rate is the rate at which the display updates when integrated with the RAMSS real-time aircraft models.

During development, a few lines of code are inserted into each display to calculate the display update rate and output the result to the screen. After the display is tested and immersed into the RAMSS simulation, the update rates are recorded. The displays are executed at their maximum capable update rate. Running the displays at their maximums is not recommended during operational activities. However, for testing and analysis purposes, allowing the displays to update as fast as possible provides a good performance measure of displays designed using a particular development tool. In the operational environment, the displays will be throttled to appropriate levels (i.e. somewhere between 16Hz and 30Hz, depending on development requirements).

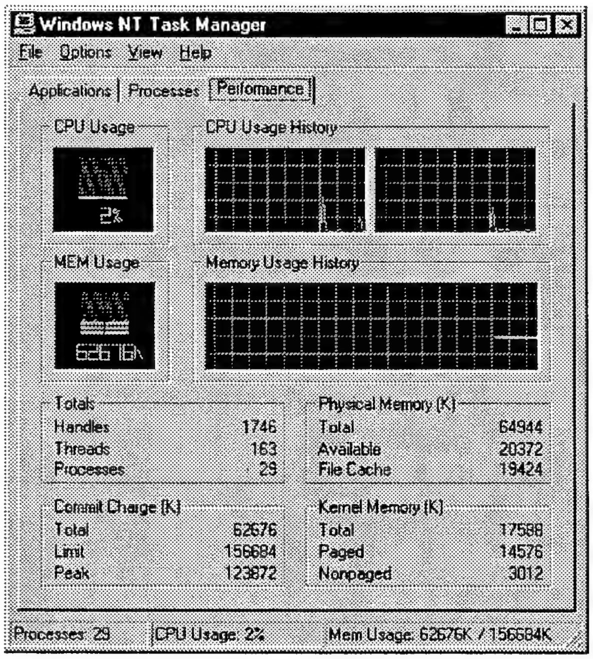
5.1.8 CPU and Memory Usage

The concepts of CPU and memory utilization are directly related to the update rate of the display. In general, as the display update rate decreases, the CPU and memory utilization decrease as well. Each display designed using OpenGL, VAPS, and Display Editor has CPU and memory usage characteristics. These are measured using software provided under the Windows NT operating system.

In the cockpit simulation environment, cockpit displays cannot hinder the performance of the aerodynamic aircraft models (RAMSS SIMPAC models in this case). The cockpit displays, while maintaining real-time performance, must use minimal amounts of the CPU and memory. The amounts that the displays actually use are highly dependent on the complexity of the display. The SIMPAC aero-models, running with an operational flight program (OFP), currently use 41% of the CPU processing power on a dual Pentium

500-MHz machine and 35% of the available memory. Through consultation with the personnel at the Advanced Architecture and Integration Branch and at SCS Engineering, Inc., it was determined that the CPU and memory usages for cockpit displays should not exceed 5%. This number allows multiple displays to execute concurrently, room for future upgrades to the models, and more complex flight programs. Non real-time displays, like a general status page, that has no real-time requirements, will generally use less CPU time and memory than real-time displays.

CPU and memory utilization are difficult to see and measure. However, administrative tools are available that allow CPU and memory usage to be monitored and quantified. One such program, provided by the Microsoft Windows NT operating system and seen in Figure 20, outputs the CPU and memory usages for the current state of the system.



```
Totals                      Physical Memory (K)
Handles         1746        Total          64944
Threads          163        Available      20372
Processes         29        File Cache     19424

Commit Charge (K)           Kernel Memory (K)
Total          62676        Total          17588
Limit         156684        Paged          14576
Peak          123872        Nonpaged        3012

Processes: 29   CPU Usage: 2%   Mem Usage: 62676K / 156684K
```

Figure 20. Windows NT Task Manager

THIS
PAGE
IS
MISSING
IN
ORIGINAL
DOCUMENT

(Synchronous DRAM). This memory, coupled with a single 300MHz Power PC processor for each display, also maintains graphics and operating system software in addition to display information [13:16-18].

Size and number of executables required for a display are important for this part of the experiment. A display built into a single self-contained executable requires less context-switching than with a display built using multiple executables. Context switching between executables consumes precious time that can mean the life or death of a pilot.

5.1.10 Display Development Time

The amount of time spent in developing a display can be crucial in the simulation environment due to time constraints and deadlines. Several issues factor into the display development time. The choice of development tool is only one of them. Other items that influence the time it takes to develop a display include personal experience with a toolset, experience with cockpit displays, general programming experience, and display complexity. In addition, two of the subjective criteria examined in this experiment, learning curve and the user interface, influence the development time as well. Having said this, it is difficult to get an accurate measure of the time it takes to develop a display using the different development tools. For this research effort, the display development time is measured and recorded upon completion of a display built with each development toolset. A short development time, for example, is advantageous in the simulation environment allowing many different missions to be run in a short period of time. Experience levels, display complexity, and tool learning curve affect the recorded numbers, however, measuring the development time gives a general idea of the time it takes to develop certain types of displays.

5.2 Experiment #2 – Post-Development Display Performance

As briefly mentioned in the introduction of this chapter, this experiment examines the displays and their performance characteristics when running in different hardware configurations. Introduced in Chapter 3, the data shown in Table 8 shows the three hardware configurations used to test the displays developed in OpenGL, VAPS, and Display Editor.

This experiment further classifies each development tool with its potential hardware dependencies. Each display developed is run on the three hardware platforms of Table 8 under the same operating conditions and the update rate of each is recorded. Running each display in this manner allows for hardware influences on display performance to be analyzed. Hardware influences such as the amount of system main memory,

Table 8. Three Hardware Configurations for Display Execution

Configuration		Processor	Memory (RAM)	Graphics Processor	Graphics Memory	Cost (Year)
1	Micron	Pentium 200-MHz	64MB	ATI Rage Pro Turbo	8MB	\$2400 (1997)
2	HP Server	Dual Pentium 333-MHz	256MB	ATI Rage Pro Turbo	8MB	\$4500 (1998)
3	HP Kayak	Dual Pentium 500-MHz	400MB	HP FX-6 Video	18MB	\$12500 (1999)

amount of graphics memory, number and speed of CPUs, type of graphics card, etc. have a large impact on display performance. The different hardware devices within each platform range in price as well. Results from this experiment will help in choosing appropriate cockpit display hardware to meet performance and budget constraints.

5.3 Summary and Other Considerations

License and hardware costs are also important in making the choice of display development toolsets. Such items, however, factor into the analysis in Chapter 6. The cost of the license to use the software is pretty much black and white. It is slightly more difficult to look at the cost of required hardware. There are literally hundreds of hardware options available for each of the development tools for this research effort. The amount of memory, number of CPUs, and graphics cards, all affect display development. For this reason, the hardware cost for the target platform is recorded (HP Kayak in Table 8), as well as the other two hardware configurations in Table 8. These costs contribute to the analysis in the following chapter.

VI. Results and Analysis

6.0 Introduction

This chapter presents the results of the experiments discussed in the previous chapter and an in-depth analysis of these results. The results are presented separately for each experiment with a summarizing analysis for each. The experiment results and analysis are followed by an all-encompassing summary discussing the development tools and their displays, advantages, disadvantages, and performance characteristics. Concluding the chapter is a brief synopsis of comments from Mr. Jesse Blair, a former Air Force pilot, crewstation graphics and display expert, and current Team Leader in the Advanced Architecture and Integration Branch.

One important factor in this research is the sample size. Due to time and availability factors, the researcher alone answered the questionnaires with direct support from the staff at the Advanced Architecture and Integration Branch. With this added support and direction, the answers to the questionnaires provided a solid foundation for the development tool analysis.

6.1 Results and Analysis of Display Development Experiment

Chapter 5 discussed two experiments, the first of which contained several questionnaires and tests about the display development lifecycle for OpenGL, VAPS, and Display Editor. Individual sections are dedicated to the results for each questionnaire, presented in tabular format, containing the answers to each question and a total score for each criterion. Following each questionnaire table is an in-depth analysis of the results. It is important to note that most of the results for the first experiment are subjective in nature and may be different for other display designers. However, the extent of the questions and the number of questionnaires should provide a solid foundation for analyzing cockpit display development software consistent with the methods used in this research effort. The results for the maintenance, update rate, CPU and memory usage, executable sizes, and development time tests are combined into a single table followed by an analysis. The same is true for the update rate, CPU usage, and memory usage tests. These objective measurements, coupled with the subjective criteria results, form a solid basis for evaluating the development potential for OpenGL, VAPS and Display Editor.

6.1.1 Learning Curve

Table 9 contains the answers to the questions pertaining to the learning curve associated with OpenGL, VAPS, and the Display Editor development toolsets. While the questions concern the influences on the learning curve, the results conclusively reveal that the OpenGL and VAPS development environments have a learning curve advantage.

One advantage that the OpenGL environment has over the other tools comes primarily from the fact that the tool is so widely used. There are countless examples available in many different application categories. Furthermore, OpenGL is widely considered as the graphics programming standard. For this reason, more and more companies are ensuring that their applications and graphics hardware are OpenGL-compliant. There are many books, papers, and tutorials available from people who have used OpenGL as well as a detailed programmer's guide (*OpenGL Programmer's Guide – Second Edition*). The single major disadvantage of OpenGL is the knowledge of software programming that is required to program by hand without a user-friendly point-and-click environment like that provided by VAPS and Display Editor.

The VAPS development tool has a couple of advantages over OpenGL. The VAPS delivered software comes with a very detailed user's manual and associated object library. Actually there are two manuals, one for the development environment (Object Editor, Stateforms Editor, and Runtime Environment) and the other for the code

Table 9. Learning Curve Questionnaire Results for Development Tools

	Question	OpenGL	VAPS	Display Editor
1	Is a user's manual/documentation available?	3	5	2
2	Is the documentation understandable?	4	3	4
3	Are examples included in software deliverable?	2	4	1
4	Are examples available elsewhere (i.e. web)?	5	2	2
5	Are examples complete and well documented?	4	4	2
6	Are courses available for the toolset?	4	4	0
7	Does analyst/designer have display experience?	3	3	3
8	Does analyst/designer have coding experience?	4	4	4
9	Is knowledge of coding NOT needed?	0	5	1
10	Do industry leaders use the development tool?	5	3	2
11	Is the software used for other applications?	5	2	0
12	Is help available online?	4	2	1
13	Is help available through software producer?	3	4	5
0 = Worst 5 = Best	Score :	46/65 (71%)	45/65 (70%)	27/65 (43%)

generation tool. In addition, the technical staff at Virtual Prototypes, Inc. is almost always available to answer technical questions, something that OpenGL lacks. VAPS, however, lacks a vast database of examples present in OpenGL.

The Display Editor development toolset has no real advantages over the other two tools with one exception. Since the tool is being developed for the Advanced Architecture and Integration Branch's simulation system, the small staff at SCS Engineering, Inc. is available to answer technical questions and even make small changes to the tool or interface to accommodate a desired feature. Because the Display Editor is in its infancy, and yet unreleased as a commercial toolset, development examples are not available. A users manual is currently in the works, but for this research effort, one did not exist. This situation caused display development problems reflected in the development times using the Display Editor environment.

6.1.2 User Interface

The data in Table 10 reveals the results of the user interface questionnaire. For this test, the interfaces for OpenGL, VAPS, and Display Editor were examined throughout display development. Answers to the questions reveal that the VAPS development environment has the clear advantage over the other tools for its user interface.

The results for this criterion are clear. The VAPS development toolset has a detailed graphical user interface (GUI) that takes advantage of a point-and-click windowed environment giving it an advantage over OpenGL. In addition, VAPS contains many predefined objects and library functions that allow the display designer to group objects together to form such things as dials, switches, lights, buttons, and potentiometers (to name only a few).

The Display Editor, while including a graphical user interface, lacks an object library forcing the display designer to manually manipulate code to achieve the desired functionality of a graphical object (dial, tape, button, etc.). However, since Display Editor is OpenGL-compliant, it uses the same online help engine as OpenGL. This help engine, provided by Microsoft Visual Studio Version 6.0, is highly detailed in C++ programming techniques, but lacks OpenGL-specific help. The help engine provides numerous C++ examples and in-depth descriptions of C++ functionality, but nothing OpenGL-related. OpenGL has no user interface except that which is provided through the C++ compiler (Visual Studio 6.0). It also has no object library available. However,

Table 10. User Interface Questionnaire Results for Development Tools

#	Question	OpenGL	VAPS	Display Editor
1	Is the user interface graphical (GUI)?	**1**	**5**	**3**
2	Does tool contain standard graphical shapes (i.e. circles, squares, etc.)?	**1**	**5**	**3**
3	Are tool bars for color, size, etc. available?	**0**	**5**	**3**
4	Is there an object library (i.e. dials, tapes, etc.)?	**0**	**5**	**0**
5	Can projects be saved in graphical format?	**1**	**5**	**4**
6	Is there an online help engine?	**3**	**4**	**3**
7	Does the help engine provide examples?	**3**	**2**	**3**
8	Is the help engine detailed?	**3**	**4**	**3**
0 = Worst 5 = Best	Score :	**12/40 (30%)**	**34/40 (88%)**	**26/40 (55%)**

an object library could be created using proper object-oriented coding techniques. This object library would be strictly code-oriented and have no graphical equivalent.

6.1.3 Display Readability

Table 11 shows the results for the readability questionnaire. This test examined several features of OpenGL, VAPS, and Display Editor, such as anti-aliasing algorithms, which contribute to display readability. The results reveal that OpenGL, VAPS, and Display Editor have similar capabilities in generating readable displays. However, the VAPS toolset has a slight advantage in certain areas over both OpenGL and Display Editor.

Table 11. Display Readability Questionnaire Results for Development Tools

#	Question	OpenGL	VAPS	Display Editor
1	Is font anti-aliasing a tool option?	4	4	3
2	Is line anti-aliasing a tool option?	5	5	4
3	Is polygon anti-aliasing a tool option?	5	5	4
4	Is implementing anti-aliasing easy?	3	4	3
5	Does the tool have large # of fonts?	1	5	2
6	Can lines, polygons, etc. be dynamically resized?	2	5	4
7	Are the fonts scalable (i.e. do they change with window resizing)?	4	4	3
8	Are they vector fonts (i.e. do they rotate with the display)?	4	4	4
9	Are display fonts clear and readable?	5	5	2
10	Are display objects clear (distinct edges)?	5	5	4
11	Are display objects readable(visually pleasing)?	5	5	3
0 = Worst 5 = Best	Score :	43/55 (78%)	51/55 (93%)	32/55 (58%)

The data in Table 11 show that all of the tools have the ability to anti-alias fonts, lines, and polygons. VAPS has the advantage of using any one of its 34 pre-defined font styles. True type fonts, available on the World Wide Web, can also be downloaded for free and easily converted into a format recognized by the VAPS toolset using a conversion routine provided with the toolset. In addition, objects defined within VAPS (including predefined objects like dials, switches, etc.) can be resized with a simple click-and-drag of the mouse, something OpenGL lacks. OpenGL does provide a related function, the "scale" command, however, programming knowledge is required to take full advantage of its potential and it still lacks the point-and-click functionality provided under VAPS.

Readability is the single most distinctive characteristic of the displays, from the pilot's point of view. A pilot will refuse to use a display that does not have clear, readable text and objects with clear edges that are visually pleasing. The Display Editor toolset uses line drawing to create the text seen on a display. For this reason, the text appears blurry and unclear during execution, resulting in a score of 2 for font clarity and readability. The OpenGL displays use predefined stroke fonts to implement the text allowing for clear anti-aliased text. The results of this test show that no one tool has a clear advantage as far as capability, however, text on both the VAPS and OpenGL displays was clearer and easier to read.

6.1.4 Display Portability

Table 12 contains the answers to the questions pertaining to portability of the displays generated by the three development toolsets. The questions concern characteristics that contribute to display portability including such things as number and size of executables, recompilation requirements, and licensing issues. Results for this criterion show that OpenGL has a portability advantage.

The single most distinguishing item in Table 12 is licensing issues. OpenGL was defined and developed by Silicon Graphics free of charge. The only license fees charged by Silicon Graphics are for source code purchases, which are unnecessary in almost all OpenGL graphics applications, including the display development in this research. Both VAPS and Display Editor require license purchases for their use. Display Editor uses a per-machine fee and VAPS is a per-user fee. This means that the licenses for Display

Table 12. Display Portability Questionnaire Results for Development Tools

#	Question	OpenGL	VAPS	Display Editor
1	Can the display be built as a single executable?	5	2	5
2	Is executable generation simple?	4	2	4
3	Are a small number of support files required?	4	3	3
4	Can the display be ported without recompiling?	4	4	4
5	If recompiling required, are code changes simple and straightforward (if not, enter 5)?	5	5	5
6	Are licensing issues avoided for portability?	5	2	2
0 = Worst 5 = Best	Score :	27/30 (90%)	18/30 (60%)	23/30 (77%)

Editor are associated with each machine on which it is installed while a single VAPS license can be installed on several machines allowing only a single user at any one time.

The other, less distinguishing piece of information in Table 12 is that the VAPS tool requires two executables for each display. One executable is the graphical definition of the display while the second is the interface between the display and the aircraft models. In the simulation environment, having two executables is not significant as a simple batch file can be used to execute both. However, in the real cockpit environment, there is limited memory and a single context switch between executables can mean life or death for a pilot.

6.1.5 Display Extensibility

Table 13 shows the questionnaire results for the display extensibility criterion. This test examined the modularity, reusability, and the general code and comments of the displays designed with OpenGL, VAPS, and Display Editor. Recall that extensibility means that a development tool has the ability to customize, reuse objects, add new functions, and modify the behavior of a developed display. The results reveal that the VAPS tool has the extensibility advantage.

The objects within the VAPS toolset are highly reusable. A simple point and click within the graphical user interface copies an object from one display to another, maintaining the object's integrity. For example, if a predefined dial is copied into a new display, it will still be a dial, with the same properties and functionality as before (the interface will have to be re-established, as expected). In addition, properties such as the size, location, and color of the copied objects are easily changed and incorporated into the new display using the Object Editor's properties window.

Display Editor allows objects to be reused in a similar manner. However, without an established object library with predefined objects, the properties and functionality of the copied objects are lost and must be regenerated. VAPS allows a dial to be copied from one display to another, preserving the properties that define the object as a dial. Display Editor lacks this functionality but does allow the pictorial representation of the object to be copied.

With this in mind, Display Editor has a slight extensibility advantage over OpenGL because it uses a GUI and can manipulate objects at the graphical level. However, the displays developed in OpenGL (hand-coded) may have meaningful

Table 13. Display Extensibility Questionnaire Results for Development Tools

#	Question	OpenGL	VAPS	Display Editor
1	Can extensions and functions be added to display at graphical level (i.e. point-and-click)?	**1**	**5**	**3**
2	Is object-oriented programming used?	**2**	**2**	**4**
3	Is display modular (i.e. groupable and selectable objects that can be copied)?	**1**	**5**	**3**
4	Does the display contain sufficient comments?	**4**	**3**	**3**
5	Do the comments have meaning?	**5**	**3**	**2**
6	Can objects in the display be reused?	**2**	**5**	**4**
7	Is reuse of objects simple (i.e. cut and paste)?	**3**	**5**	**4**
8	Is adapting the reused object simple and straightforward (i.e. changing size, color, etc.)?	**3**	**5**	**3**
0 = Worst 5 = Best	Score :	**21/40 (53%)**	**33/40 (83%)**	**26/40 (65%)**

comments that make it easier for a display designer to cut, paste, and adapt sections of code for other display use. These comments are detailed descriptions of procedures and functions within a display that have the potential to be reused in other displays.

6.1.6 Subjective Criteria Summary

The subjective criteria scores are summarized in Figure 21. The chart shows the total score for each development toolset in the various subjective categories. The VAPS toolset has the highest average score while the Display Editor toolset has the lowest.

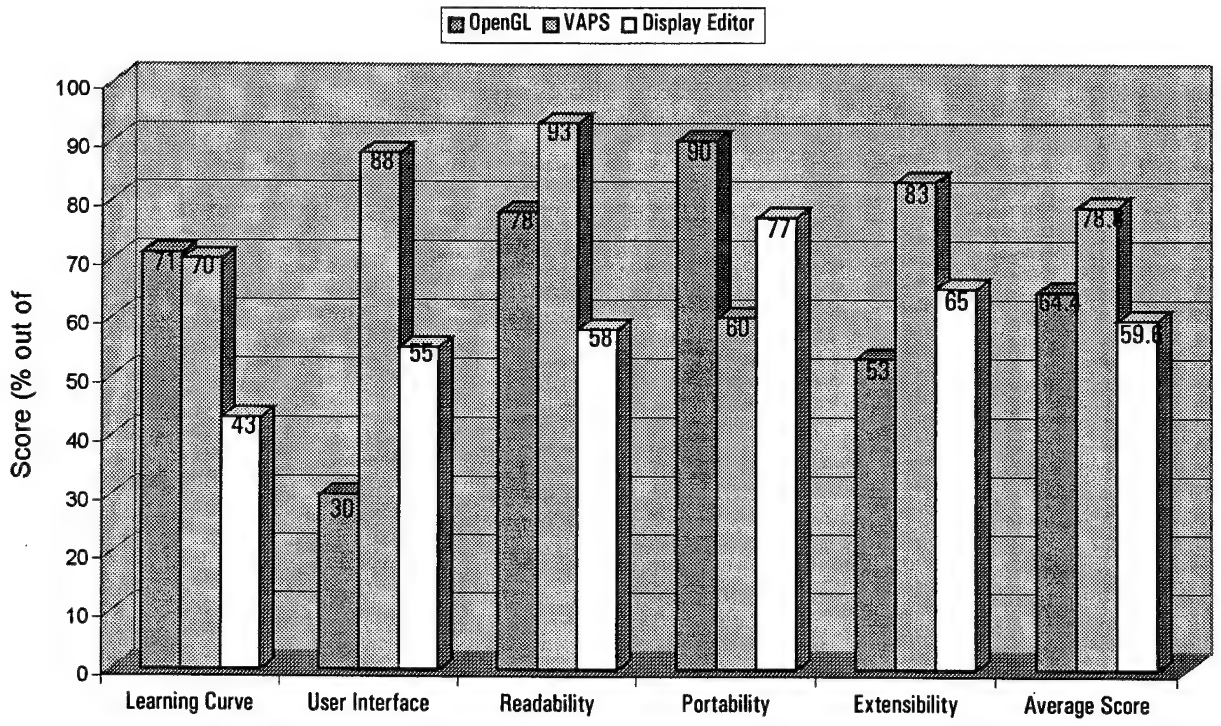


Figure 21. Summary of Subjective Criteria

From the subjective criteria perspective, the data in the chart shows that the VAPS toolset has a greater potential for successfully developing graphical cockpit displays. While the OpenGL toolset has advantages in portability and learning curve, the VAPS toolset scores consistently higher across the board. The Display Editor toolset consistently scored poorly by comparison to VAPS and OpenGL.

6.1.7 Display Development Time

Development times for each display built using OpenGL, VAPS, and Display Editor were recorded following completion of each display. Each display was built from start to finish without working on any other display. This ensured that the time spent on each display was dedicated solely to that display and that the toolset did not have to be relearned each time the tool was used. Table 14 shows the development times for each display using OpenGL, VAPS, and Display Editor.

The displays developed using the VAPS development toolset were accomplished the fastest. Hand-coding with the OpenGL toolset took the longest, while the Display Editor toolset fell in between. The Display Editor timing results include changes made to

Table 14. Development times for displays in OpenGL, VAPS, and Display Editor

Display Developed	Development Time
ADI – OpenGL	80-85 hrs
ADI – VAPS	20-22 hrs
ADI – Display Editor	40-45 hrs
HSI – OpenGL	50-55 hrs
HSI – VAPS	10-12 hrs
HSI – Display Editor	38-39 hrs

the tool throughout the display development lifecycle. As mentioned earlier, the display designer simply drew the displays, while the staff at SCS Engineering, Inc., accomplished the dynamic animation. Clearly, the results show that hand-coding a display requires much more development time than using a windowed point-and-click environment such as that provided in VAPS or Display Editor.

6.1.8 Display Maintenance

The maintenance measurements were recorded as an average time to accomplish changes or updates for the displays built using the three development environments. Types of required changes included adding a simple text field to output information or adding a switch that allowed the changing of display mode of operation. An example of this can be seen in the ADI display. It has two modes of operation; day mode and night mode. In day mode, the sky is blue, the ground is green, and lines colors are white. In night mode, the sky is gray, the ground is black, and line colors are green. Adding a mode switch such as this was considered maintenance for this research effort. The mean-time-to-change (MTTC) numbers can be seen in Table 15.

Table 15. MTTC Ratings for OpenGL, VAPS, and Display Editor

Development Environment	Types of Changes Required	Mean-Time-to-Change Values (Average)
OpenGL	Day/Night Switch, ILS & TACAN Functions, Add Simple Text Field	(1½ + 4 + ¼)/3 = 1.9 Hrs
VAPS	Day/Night Switch, ILS & TACAN Functions, Add Simple Text Field	(½ + 1½ + ½)/3 = 0.83 Hrs
Display Editor	Color Changes, Pitch Ladder Changes, Add Simple Text Field	(1 + 4 + ½)/3 = 1.83 Hrs

The VAPS toolset has a clear advantage over the other two toolsets. The ability to use the VAPS point-and-click windowed environment to make all necessary changes reduced the system down time significantly. The OpenGL environment required hand coding of the changes. The current state of the Display Editor environment required the changes to be made in both its point-and-click code generation environment and its manual OpenGL programming interface. The data from Table 15 shows that the point-and-click environment provided by the VAPS toolset reduces maintenance down time due to upgrades or changes to the displays. These numbers correlate nicely with the results seen for the development times in Table 14 above.

6.1.9 Executable Size

Each display in the F-22 Raptor has limited memory (32MB SDRAM – no hard drive) for storage of the display executable, the graphics processing software, and the operating system software [13:16-18]. With this in mind, the displays developed for this research effort were examined for executable size and runtime memory usage. Table 16 shows the sizes for the executables required for the displays while the following sections covers the runtime memory usages.

The OpenGL environment has a clear advantage in executable size. The OpenGL executables in Table 16 are nearly 10 times smaller than those produced using Display Editor or VAPS. In fact, the VAPS toolset requires a second, smaller executable to

Table 16. Executable Sizes for OpenGL, VAPS, and Display Editor

Display Developed	Number of Required Executables	Executable Size
ADI – OpenGL	1	137K
ADI – VAPS	2	1.95MB + 494K
ADI - Display Editor	1	1.45MB
HSI – OpenGL	1	205K
HSI – VAPS	2	1.85MB + 495K
HSI – Display Editor	1	1.45MB

interface between the simulation environment and the display, adding to its overall executable size. As is common with code generators, both VAPS and Display Editor generate large amounts of software code. Since they use a point-and-click environment, the VAPS and Display Editor code generators must incorporate required point-and-click functions associated with the generated code. When compiled and linked, these large amounts of code equate to larger executables, as seen in Table 16.

6.1.10 Update Rates, CPU Usage, and Memory Usage Results and Analysis

As noted in Chapter 5, the display update rate is one of the most important criteria by which a display is measured. Due to the limited space and processing power in the F-22 cockpit, the CPU and memory usages are also important display criteria. To get an accurate measure of the maximum display performance, each was executed while embedded in the RAMSS simulation environment and allowed to run open loop. In the case of the OpenGL and VAPS development environments, the update rate was measured using a small piece of software code that when executed, displayed the update rate in the lower left or right corner of the display. For the Display Editor toolset, the update rate was measured using a tool provided by the RAMSS simulation environment called Monitor that output the update rate for the display on a text page. The display update rates are shown in Table 17.

The displays were then throttled down to an appropriate level for measuring the CPU usages. The data in Table 17 reflects CPU usages for each display running at 16Hz and 30Hz update rates. The memory usage was measured using the 30Hz-throttled displays. All CPU and memory usages were measured using the Windows NT Task Manager (see Figure 20) and then recorded.

All of the displays meet the real-time update rate requirements established in the standards document. In fact, all of the displays built using OpenGL, VAPS, and Display Editor nearly quadruple the minimum 16Hz requirement. The displays built by hand using the OpenGL environment have the smallest CPU and memory usages while the VAPS toolset has the largest usages. The displays built with smaller executables (OpenGL) tend to use less of the CPU during execution (1-2%). Since VAPS uses two larger executables during execution, a larger CPU usage is seen (2-4%).

Table 17. Update Rate in Open Loop Configuration

Display	Update Rate (Open Loop)	CPU Usage (30Hz)	CPU Usage (16 Hz)	Memory Usage (30Hz)
ADI – OpenGL	93.2	2%	1%	3408 Kb
ADI – VAPS	52.9	4%	3%	5176 Kb
ADI – Display Editor	59.9	6%	5%	4936 Kb
HSI – OpenGL	59.2	2%	1%	3808 Kb
HSI – VAPS	52.8	3%	2%	4908 Kb
HSI – Display Editor	51.3	7%	5%	4264 Kb

6.2 Hardware Configuration Experiment

The second experiment executed the displays in several hardware configurations. Table 18 reviews the hardware configurations used for this experiment. The displays were run in an open loop configuration to establish the maximum display performance in each hardware platform in the table. The update rates for the displays were then recorded for each configuration.

One important note is that the HP Kayak machine (#3 in Table 18) contains an HP FX-6 graphics card, which has built-in OpenGL acceleration on-board the graphics processor. Figure 22 shows the update rates for each display in each hardware configuration from Table 18.

These results clearly show that OpenGL acceleration is an absolute must when using OpenGL or Display Editor. The HP Kayak, with its OpenGL acceleration, far outperforms the other two hardware configurations. The HP Server, containing more memory, more CPU power, and the same graphics card as the Micron, performs only slightly better. From the results seen in Figure 22 it is concluded that the ability of the

Table 18. Hardware Configurations for Update Rate Experiment

Configuration		Processor	Memory (RAM)	Graphics Processor	Graphics Memory	Cost (Year)
1	Micron	Pentium 200-MHz	64MB	ATI Rage Pro Turbo	8MB	$2400 (1997)
2	HP Server	Dual Pentium 333-MHz	256MB	ATI Rage Pro Turbo	8MB	$4500 (1998)
3	HP Kayak	Dual Pentium 500-MHz	400MB	HP FX-6 Video	18MB	$12500 (1999)

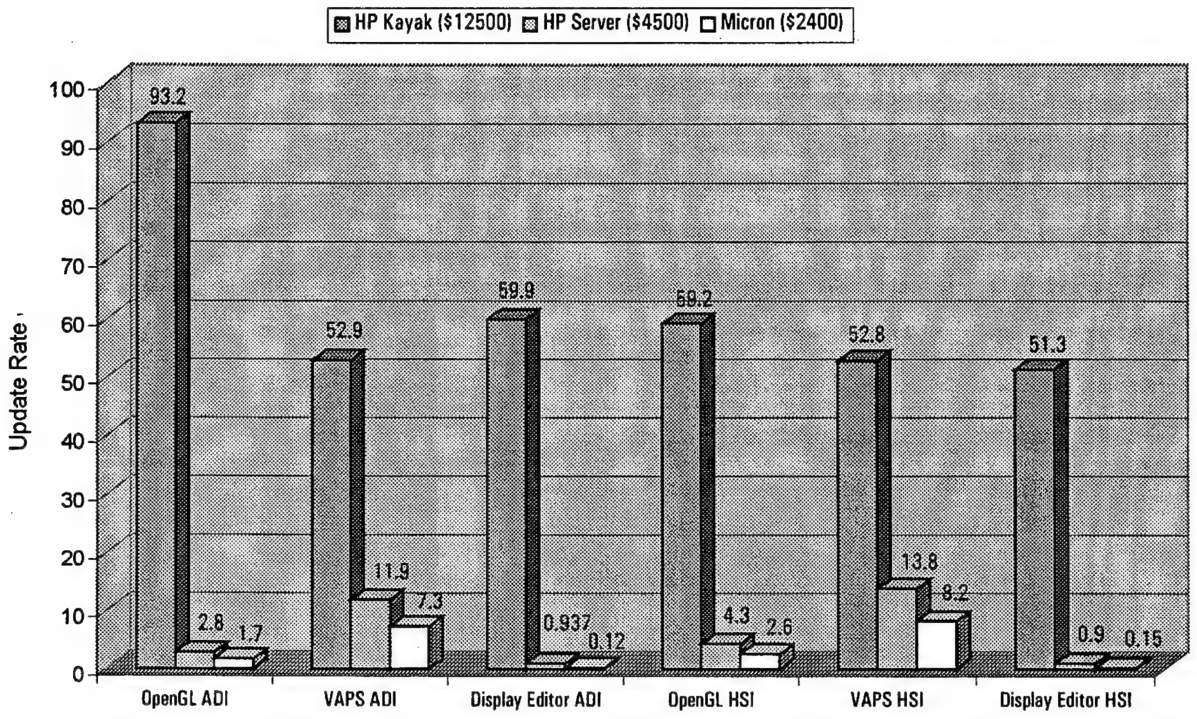


Figure 22. Update Rates for displays in each hardware configuration from Table 18

graphics processor is the dominant factor. While more memory and CPU power improve performance slightly, adding an OpenGL graphics accelerator increased performance by a factor of nearly 5 for VAPS, 10+ for OpenGL, and 50+ for Display Editor.

6.3 Analysis Summary

The experiments conducted during this research effort revealed the advantages and disadvantages of the OpenGL, VAPS, and Display Editor cockpit display development tools. Table 19 summarizes the results for all of the criteria in the first experiment.

Table 19. Summary of Development Tool Results for Cockpit Displays

Category		Display Development Toolset		
		OpenGL	VAPS	Display Editor
1	**Learning Curve**	A	A	N
2	**User Interface**	D	A	N
3	**Readability**	A	A	N
4	**Portability**	A	D	N
5	**Extensibility**	D	A	N
6	**Development Time**	D	A	D
7	**Executable Size**	A	D	D
8	**Maintenance**	D	A	N
9	**Update Rate**	A	A	A
10	**CPU/Memory Usage**	A	N	D
	Advantages:	6	7	1
	Disadvantages:	4	2	3
	Neutrals:	0	1	6

Based on the data in the chart, the VAPS toolset has the best advantage to disadvantage ratio at 3.5. The OpenGL toolset has a higher ratio (1.5) than that of the Display Editor software (0.33). These results reflect the applicability of each software development toolset in developing cockpit displays. Other designers using these tools should weight their results according to the application for which they are being used.

The Display Editor's primary disadvantage is simply its immaturity. Technically not yet released to the public, the toolset lacks functionality and features that are normally present in similar software tools like the VAPS toolset. Upon initial release to the public, the tool will more than likely change some of the neutrals in Table 19 to advantages and increase its advantage to disadvantage ratio. As a distinct advantage, SCS Engineering, Inc. is easily contacted and extremely helpful in clarifying Display Editor components and features making the toolset easier to use.

The VAPS toolset's largest drawback is the licensing fees. At roughly $35K per license, the VAPS tool is very expensive. However, the added benefits of the VAPS toolset more than make up for the high cost. Using a windowed point-and-click user interface, VAPS allows for easy display creation, quick maintenance, and painless extensions. Also, having a built in library of defined objects makes creating cockpit displays simple and straightforward.

The OpenGL tool's biggest drawback is the requirement of programming knowledge. If a user has no programming knowledge, the learning curve becomes lengthy. OpenGL has a broad user base and countless examples available throughout the industry, providing any non-programmer with example code to speed learning. Furthermore, OpenGL has a distinct portability advantage over the other toolsets as described in the Display Portability section with its many potential operating environments.

The test across varying hardware configurations revealed one major point. The graphics card chosen to drive the graphics applications in any hardware configuration is crucial. The results in Table 19 show that the addition of a graphics card with on-board OpenGL acceleration provides a significant boost to performance for displays built using all three development toolsets. This point is important to understand as simply adding CPU power and memory may not achieve desired performance increases.

6.4 Analysis Summary from Research Field Expert

Mr. Jesse Blair, Display Expert, Former Pilot, and Team Leader in the Advanced Architecture and Integration Branch, reviewed the above results and analysis. This section briefly summarizes his comments and observations.

After reviewing the above results and analysis, Mr. Blair states, "After careful review of the researcher's results, it is clearly demonstrated that the graphics community is finally able to use automated code generating toolsets for both research and production aircraft avionics systems." He goes on to state that while the VAPS toolset has slightly lower performance characteristics than OpenGL, it is now an acceptable development toolset for most applications. Mr. Blair suggests, "Because OpenGL manual software generation still has many advantages over a toolset like VAPS, probably the best near term approach to crewstation display format development is to use a combination of manual OpenGL and VAPS." He continues to say that when high performance requirements need to be met, the OpenGL toolset could be used, and for all others, the VAPS automatic code generation system. He further observes that "prototyping using the VAPS toolset and then using manual OpenGL software generation only where performance dictates seems like a logical conclusion from this research."

VII. Summary, Conclusions, and Future Work

7.0 Summary

This research effort had three primary objectives. First was developing the displays using OpenGL, VAPS, and Display Editor. Using the standards and requirements established by the F-22 Raptor design team at Lockheed-Martin and Boeing, the three development toolsets were used to build both an Attitude Director Indicator and Horizontal Situation Indicator. Meeting the first research objective ensured that OpenGL, VAPS, and Display Editor were each able to meet the ADI and HSI display standards and requirements established in the F-22 Standards Document [4]. Had one of the tools been unable to meet the requirements, it would have to have been ruled out of any analysis comparison, degrading the remaining research objectives.

The second objective established a foundation for analyzing development tools and the displays built using them and provided evaluation techniques that can be used in analyzing future cockpit displays and development toolsets. The questionnaires established in Chapter 5 went into detail on how a development toolset meets the requirements and standards for a display. In addition, examining key objective criteria allows for further characterization of development toolsets. While these techniques are applied specifically to the display development arena, they could be adapted to evaluate any general graphics package by massaging the questions slightly to broaden the scope of the evaluation.

The final objective determined the performance characteristics of OpenGL, VAPS, and Display Editor and their cockpit display development potential. While this research did not determine the better of the three toolsets, it was clear in assigning advantages and disadvantages to the three toolsets with respect to cockpit display development. It would be inappropriate to compare the new and unreleased Display Editor toolset with two of today's well-established cockpit graphics toolsets, OpenGL and VAPS. A comparison, revealing the advantages and disadvantages of the three toolsets, allows future display designers to weigh these advantages and disadvantages when developing graphics applications for the cockpit or elsewhere.

7.1 Relationship to Past Efforts

Like the AGSSS discussed in Chapter 2, these three development tools significantly reduce the lifecycle of display development. While OpenGL still requires extensive programming experience, the number of examples available provides inexperienced programmers with needed tips, tricks, and tools. The VAPS and Display Editor environments take key concepts from early efforts, like the AGSSS development tool of the early 1980's. Both tools provide a windowed, point-and-click environment similar to the AGSSS, but much more extensive and user-friendly. The VAPS and Display Editor environments allow for faster, easier, and more cost effective creation of cockpit displays than conventional hand coding.

7.2 Conclusions on Display Design Methodology

This research effort focused on developing cockpit displays using a linear sequential design model. The five steps of the linear sequential model, shown in Table 2, were easily adapted to the development of cockpit displays. The requirements analysis document from the first phase of the model equated directly to the F-22 standards document created by the F-22 design team at Lockheed-Martin and Boeing. The function, behavior, performance, and all interfacing requirements were established in the F-22 standards document and applied to each display during development.

The design phase of the linear sequential model used the Requirements Analysis document (F-22 standards document) to develop the necessary data structures and interfaces that would be used in the displays. The design phase for this research effort was fairly simple as the displays built were typically a single piece of software (i.e. only one or two executables with limited interfacing requirements). The design phase was important, however, in determining the appropriate interface variables from the simulated aircraft models (RAMSS).

The code generation phase was very simple using the generators provided under VAPS and *Display Editor*. These two tools use the point-and-click environment to create the graphical displays and allow the corresponding code to be generated with the simple click of a mouse button. Using the OpenGL toolset, on the other hand, required the entire display to be coded by hand, lengthening the OpenGL code generation phase significantly.

The testing phase entailed developing a test routine that would exercise the display. In each environment, this routine sent generic data to each display to test the look and functionality of the display ensuring that display objects were within operating parameters. This phase is separate from the research objective that immerses the displays in the RAMSS simulation environment. The testing phase of the linear sequential model is used to simply test the displays for correct operation and function using generic data.

Finally, the maintenance phase of the model is still in effect. Once developed, the displays required changes resulting from errors or added functionality requirements. While some of these changes were accomplished for the maintenance analysis, others will be implemented in later maintenance efforts. The maintenance of software deliverables is an ongoing process, which has only just begun for the display developed in this research effort.

7.3 Conclusions from Comparative Analysis

The comparative analysis accomplished in Chapter 6 revealed numerous advantages, disadvantages, and performance characteristics of OpenGL, VAPS, and Display Editor. Some readers may make conclusions that one tool is better than the others based on the analysis results, while other readers may conclude that a different tool is the best. The analysis results allow a display designer to examine the characteristics of a development toolset and determine if it is the right tool for their needs. Since different development facilities have vastly different requirements, one tool may not suit one facility while it may be the perfect toolset for another. The comparative analysis reveals the necessary characteristics of OpenGL, VAPS, and *Display Editor* allowing the best possible choice for a display design team.

Though the VAPS toolset has a number of advantages that the other tools do not have, it is not without its limitations. VAPS was designed to accomplish displays that are currently being used in today's aircraft. This presents a problem to facilities and contractors that wish to create highly complex and advanced displays. For example, developing a display in 3-dimensional space is impossible in VAPS because it simply is not a design feature. VAPS is designed around current cockpit instrumentation (i.e. tapes, dials, buttons, lights, etc.). OpenGL, on the other hand, is highly generalized and has the ability to be programmed in 3-dimensional space.

7.4 Conclusions on Evaluation Techniques

The analysis conducted in Chapter 6 involved only three graphics packages and their capabilities for building cockpit displays. The evaluation techniques used, however, can be applied to virtually any graphics packages in a wide variety of capabilities. Several of the evaluation techniques center on specific cockpit display characteristics but could easily be adapted to a general display/scene characteristic. The readability criteria, for example, could be applied to any type of graphics application that requires the resulting graphical display or scene to be clear and readable. The questionnaires for learning curve and user interface could be used directly in evaluating other graphics packages such as Multigen or Corel Draw. The evaluation techniques are not all inclusive. There are other points of interest that could have been examined such as toolset simplicity or feedback. The evaluation techniques used in this research effort have a direct impact on the development toolset chosen to build cockpit displays.

7.5 Future Work

Several items of interest have arisen from this research effort. Since the Display Editor is a new development toolset and will mature with time, a future analysis may reveal enhancements to the tools abilities. OpenGL and VAPS are mature and well established in industry and will continue to grow and evolve, however, Display Editor has greater room for growth because of its immaturity.

This research effort focused on only two F-22 cockpit displays. There is potential for research into other F-22 displays as well as other aircraft cockpit displays. There is also potential research into the Head-Up type displays for both the cockpit environment and the simulation environment. One display that has yet to be examined in detail is the idea of a Moving Map display. The potential for OpenGL, VAPS, or Display Editor to develop an F-22 Moving Map display is under consideration, but as of yet, no research or development has begun.

Finally, the need for graphical software development is clear. Most engineers have little programming experience and desire tools that do not require extensive coding experience. Since OpenGL is rapidly becoming an industry standard, research into a windowed, point-and-click, OpenGL code-generating development environment has only just begun. Companies like SCS Engineering, Inc, with their Display Editor, are venturing into OpenGL-based windowed environments to facilitate graphics software development.

Bibliography

1. Bennett, Paul A. "Applications of Display Prototyping and Rehosting Tools to the Development of Simulator, Flight Training Device, and Cockpit Displays". American Institute of Aeronautics and Astronautics, 1996.

2. Bennett, Paul A. Rapid Development and Rehosting of Dynamic Graphical Interfaces. Montreal, Quebec, Canada: Virtual Prototypes, Inc. October 1997.

3. Edwards, R. J. G. "The Presentation of Static Information on Air Traffic Control Displays," Proceedings of Advisory Group for Aerospace Research and Development (AGARD) no. 255. London: Technical Editing and Reproduction Ltd, 1980.

4. "F-22 Air Vehicle Cockpit Design Description Document," CDRL-A017. OT-90-34230. Contract F33657-91-C-006 with Lockheed-Martin. July 1996.

5. Krell, Tim A. "Perils of the Glass Cockpit: The Human Factor of Computer Graphics." CS-360 Mini Report. http://www.npl.com/~tkrell/writings/aviation/glass-cockpit.html. February 1997.

6. Montoya, R. Jorge, et al, AGSSS: The Airborne Graphics Software Support System. WL-TR-91-1042. Wright-Patterson AFB OH: Wright Laboratory Avionics Directorate, September 1991.

7. Montoya, R. Jorge, et al, "An Interactive Graphics Editor for Computer Generated Cockpit Displays," Proceedings of the 9th Annual Digital Avionics Systems Conference. 441-446. 1990.

8. "OpenGL Frequently Asked Questions," Excerpt from unpublished article, 10 pages. http://www.sgi.com/software/opengl/faq.html.

9. "OpenGL Data Sheet," Excerpt from unpublished article, 10 pages. http://www.sgi.com/software/opengl/datasheet.html.

10. "Pilot Situational Awareness," Excerpt from unpublished article, 2 pages. http://www.brooks.af.mil/HSC/products/doc50.html.

11. Pressman, Roger S. Ph.D. Software Engineering: A Practitioner's Approach. New York: The McGraw-Hill Companies, Inc. 1997.

12. Sullivan, Wilf and Bennett, Paul A. COTS Products Provide Rapid Development and Deployment of Avionics and Vetronics Displays. Katana, Ontario, Canada: DY 4 Systems, Inc. July 1996.

13. "VAPS Users Final," F-22 'Raptor' Air Dominance Fighter. CD-ROM. Avionics Analysis and Integration, Advanced Development Lab, Lockheed-Martin. 1999.

14. Way, T.C., 3-D Imagery Cockpit Display Development. WRDC-TR-90-7003. Wright-Patterson AFB OH: Cockpit Integration Directorate, August 1990.

15. Woo, Mason and others. OpenGL Programming Guide: Second Edition. Reading, Massachusetts: Addison Wesley Longman, Inc. 1997.

References

1. Bailey, D. C. "F-22 Cockpit Display System," Proceedings of the SPIE Conference. 157-165. Washington: SPIE-The International Society for Optical Engineers, 1994.

2. Barbato, Gregory J. and Boucek, G. Scott. Integrating Cockpit Technologies to Improve Aircrew Situational Awareness and Performance in Ground Attack Mission. Wright-Patterson AFB OH: Vehicle-Pilot Integration Branch, 1996.

3. Benning, Stephen L., et al, Pave Pillar In-House Techinical Report Demonstrations I and IA. WRDC-TR-90-1157. Wright-Patterson AFB: Avionics Laboratory, July 1990.

4. Braaten, Alan J. A Graphics Environment Supporting the Rapid Prototyping of Pictorial Cockpit Displays. MS Thesis, AFIT/GCS/MA/86D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986 (AD-A178636).

5. Integrated Test Bed, Advanced Architecture and Integration Branch, Interface Control Document for the Mission Operation Sequence. GSTB-1280228. Wright-Patterson AF OH: Air Force Research Laboratory, 1994.

6. Jauer, Richard A. and Grasso, Joseph A. Graphics Processor Definition II. WL-TR-91-7003. Wright-Patterson AFB OH: Wright Laboratory, June 1991.

7. Kanko, Mark A. Geometric Modeling of Flight Information for Graphical Cockpit Display. MS Thesis, AFIT/GCE/ENG/87D-6. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A190484).

8. Veitch, William A. and Atkinson, Robert. An Advanced Concept for Avionics Testing. Excerpt from published White Paper, 6 Pages, Virtual Prototypes, Inc. http://www1.virtualprototypes.ca/VSD/press/cst_acat.html. July 1999.

9. Virtual Prototypes, Inc., VAPS 5.1 User's Guide, Copyright 1999.

10. Virtual Prototypes, Inc., VAPS 5.1 Conceptual Overview, Copyright 1999.

11. Way, T.C., et al, Pictorial Format Display Evaluation. AFWAL-TR-84-3036. Wright-Patterson AFB OH: Flight Dynamics Directorate, May 1984.

Vita

Captain Matthew J. Gebhardt was born on July 8, 1972 in Salem, Oregon. He graduated from A. Crawford Mosley High School in Panama City, Florida in 1991 and entered the United States Air Force Academy in Colorado Springs, Colorado. He graduated in 1995 with a Bachelor's Degree in Electrical Engineering. Captain Gebhardt served three years in the Advanced Architecture and Integration Branch of the Information Systems Directorate of the Air Force Research Laboratory before entering the School of Engineering at the Air Force Institute of Technology in August of 1998. He is married to Tonya (Snyder) Gebhardt of Dayton, Ohio and currently does not have any children.

Next Military Address:	Captain Matthew J. Gebhardt 49 Spruce Street PSC 1000 Langley AFB, VA 23665-2800
Permanent Address:	809 W. 1st Street Newberg, OR 97132

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** March 2000	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE A COMPARATIVE ANALYSIS OF COCKPIT DISPLAY DEVELOPMENT TOOLS	**5. FUNDING NUMBERS** EN - if funded, enter funding number
6. AUTHOR(S) Matthew J. Gebhardt, Captain, USAF	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GE/ENG/00M-10
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFSC Attn: Mr. Jesse L. Blair 2241 Avionics Circle WPAFB OH 45433-7765 DSN: 785-4827 x3337	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

Lt Col Timothy Jacobs, ENG, DSN: 785-3636, ext. 4279

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

ABSTRACT (*Maximum 200 Words*)

Currently, no standard methodology exists that enables cockpit display engineers to evaluate software tools used in the development of graphical cockpit displays. Furthermore, little research has been accomplished in comparing current software development tools with traditional hand-coded methods. This research effort discusses a framework for analyzing cockpit display software development tools and follows through with a detailed analysis comparing today's hand-coding standard, OpenGL, with two of today's cockpit display software development suites, *Virtual Application Prototyping System* (VAPS) and *Display Editor.* The comparison exploits the analysis framework establishing the advantages and disadvantages of the three software development suites. The analysis framework is comprised of several detailed questionnaires that enable the cockpit engineer to quantify important subjective criteria such as learning curve, user interface, readability, portability, extensibility, and maintenance. The questionnaires developed for each subjective criterion contain questions with weighted answers that enable the cockpit engineer to evaluate graphical software development tools. The questions were adapted from multiple sources including personal experience, display experts, pilots, navigators, case tool, and text sources. In addition, the comparative analysis evaluates several objective criteria with respect to development tools and the displays generated with them such as update rate, development time, executable size, and CPU/Memory usage level.

14. SUBJECT TERMS Cockpit Displays, Software Evaluation, Cockpit Display Analysis, Display Editor, Interactive Graphics Editor, Airborne Graphics Software Support System, VAPS, OpenGL, Subjective Evaluation, Objective Evaluation, hand-coded, automated code generation, case tools, graphics, cockpit graphics, dynamic graphics, heads-down, process metrics, process evaluation.	**15. NUMBER OF PAGES** 118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102